US006519712B1

(12) **United States Patent**
Kim et al.

(10) **Patent No.: US 6,519,712 B1**
(45) **Date of Patent: Feb. 11, 2003**

(54) INDEPENDENT CHECKPOINTING METHOD USING A MEMORY CHECKPOINT ON A DISTRIBUTED SYSTEM

(75) Inventors: **Do Hyung Kim**, Taejon (KR); **Chang Soon Park**, Taejon (KR)

(73) Assignee: **Electronics and Telecommunications Research Institute**, Taejon (KR)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/495,587**

(22) Filed: **Feb. 1, 2000**

(30) **Foreign Application Priority Data**

Oct. 19, 1999 (KR) 99-45258

(51) **Int. Cl.**^{7} **G06F 11/00**

(52) **U.S. Cl.** **714/15**; 714/16; 714/20; 714/21; 712/228

(58) **Field of Search** 714/13, 16, 15, 714/20, 21; 712/228

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,630,047 A		5/1997	Wang	395/182.13
5,802,267 A	*	9/1998	Shirakihara et al.	714/15
5,832,201 A		11/1998	Hirayama et al.	395/182.11
5,923,832 A	*	7/1999	Shirakihara et al.	712/228
6,332,200 B1	*	12/2001	Meth et al.	707/203
6,397,352 B1	*	5/2002	Chandrasekaran et al.	709/201

OTHER PUBLICATIONS

Xu et al., "Adaptive Independent Checkpointing for Reducing Rollback Propagation," *in Proc. of the Fifth IEEE Symp. On Parallel and Distributed Processing*, 1993, pp. 754–761.

Baldoni et al., "An Indexed-based Checkpointing Algorithm for Autonomous Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems, 10*(2):181–192, Feb., 1999.

* cited by examiner

Primary Examiner—Scott Baderman
Assistant Examiner—Christopher S McCarthy
(74) *Attorney, Agent, or Firm*—SEEP IP Law Group PLLC

(57) **ABSTRACT**

An independent checkpointing method using a memory checkpoint on a distributed system that includes a message transmission routine, a message processing routine, and a periodical checkpoint routine. The message transmission routine adds a self checkpoint number to a message to be transmitted when a current process tries to send a message to another process. The message processing routine performs a memory checkpoint and processes a message in reference to a checkpoint number of a transmission process, a checkpoint number of the current process, a memory checkpoint flag, and a message transmission flag when a message is received from a process. The periodical checkpoint routine performs a checkpoint that records a necessary state information for recovery against faults periodically in reference to the memory checkpoint flag.

4 Claims, 4 Drawing Sheets

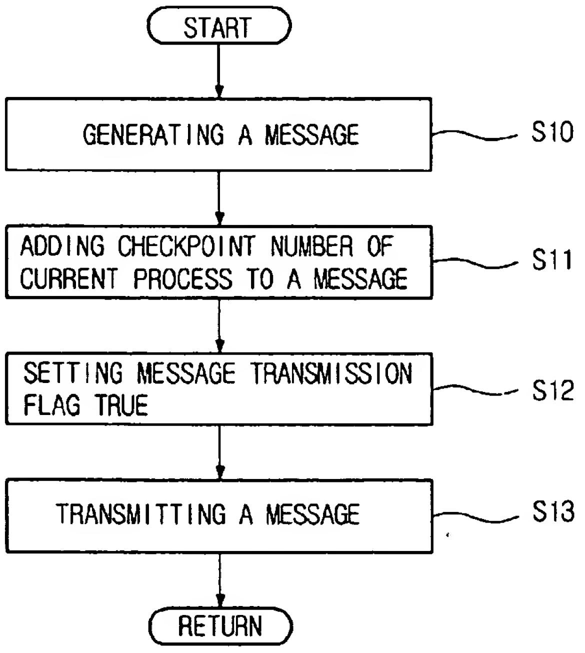

START
GENERATING A MESSAGE
S10
ADDING CHECKPOINT NUMBER OF CURRENT PROCESS TO A MESSAGE
S11
SETTING MESSAGE TRANSMISSION FLAG TRUE
S12
TRANSMITTING A MESSAGE
S13
RETURN

FIG. 1

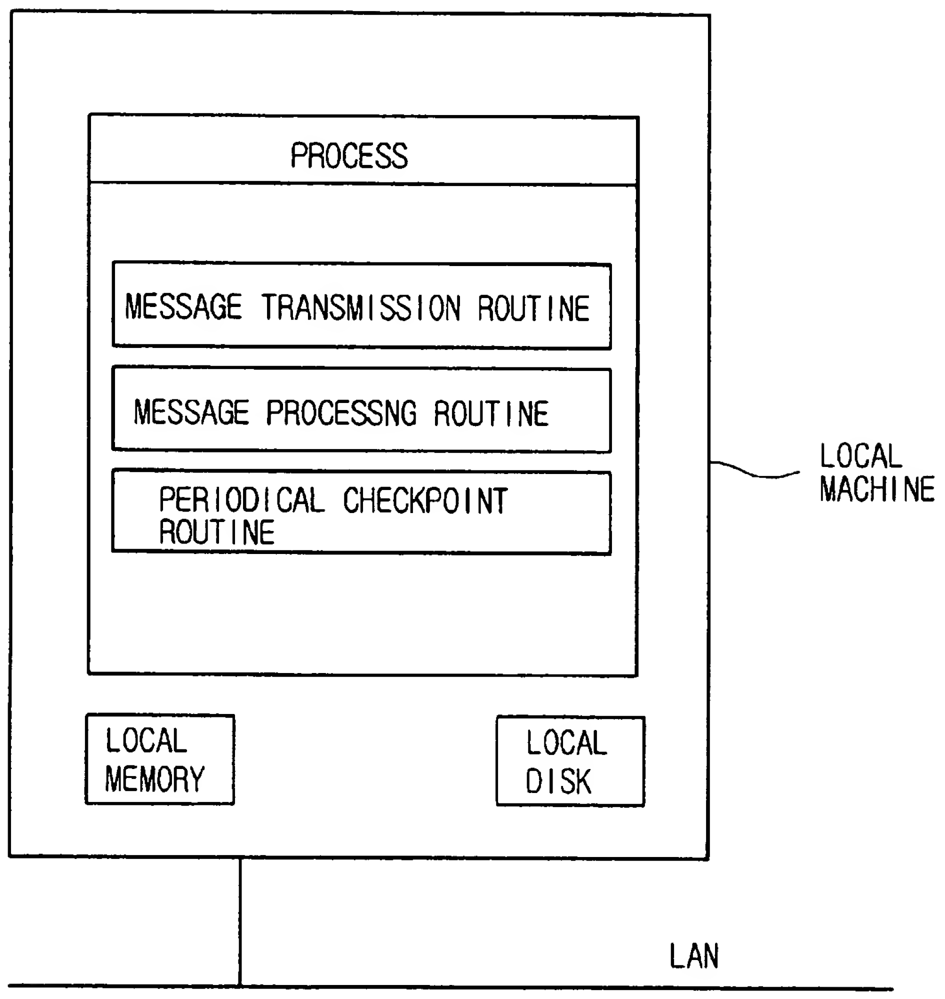

PROCESS
MESSAGE TRANSMISSION ROUTINE
MESSAGE PROCESSNG ROUTINE
PERIODICAL CHECKPOINT ROUTINE
LOCAL MACHINE
LOCAL MEMORY
LOCAL DISK
LAN

FIG. 2

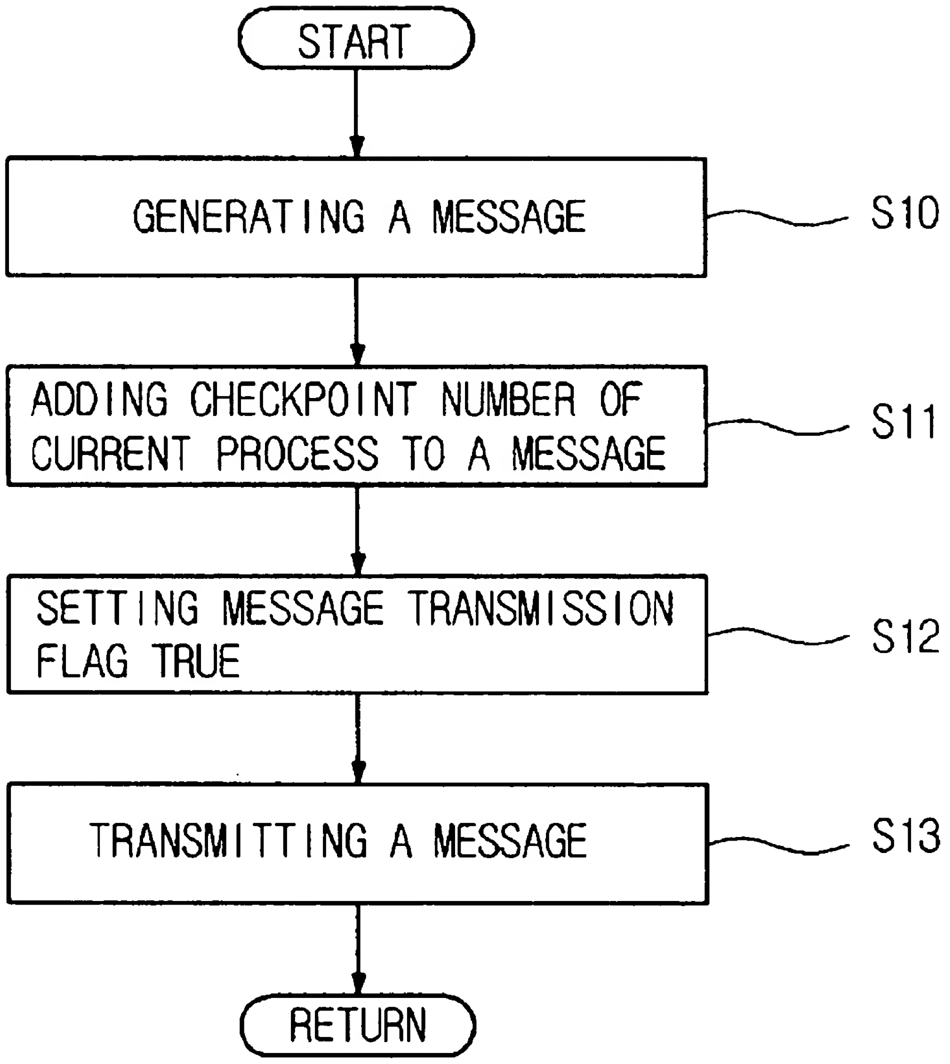

START
GENERATING A MESSAGE
S10
ADDING CHECKPOINT NUMBER OF CURRENT PROCESS TO A MESSAGE
S11
SETTING MESSAGE TRANSMISSION FLAG TRUE
S12
TRANSMITTING A MESSAGE
S13
RETURN

FIG. 3

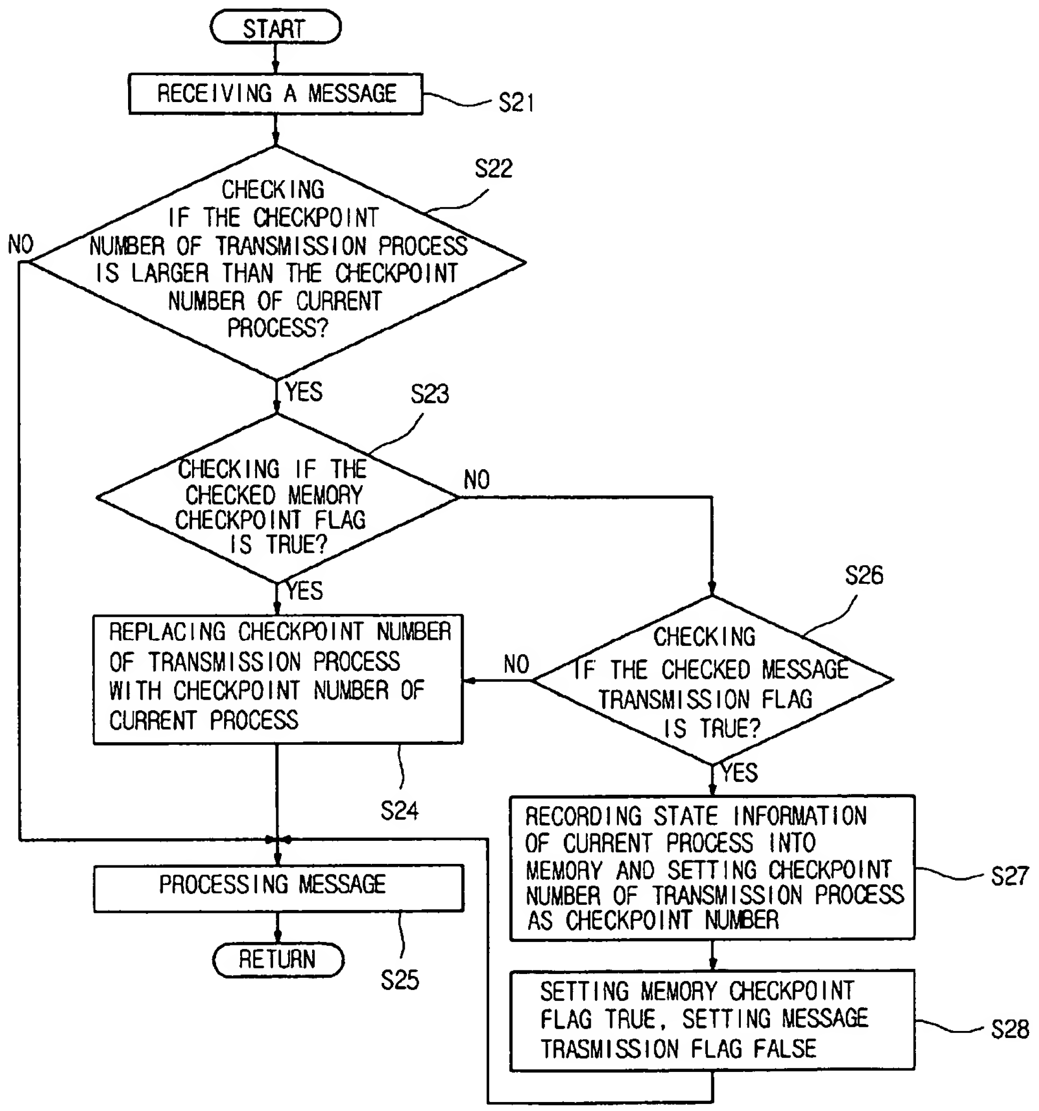
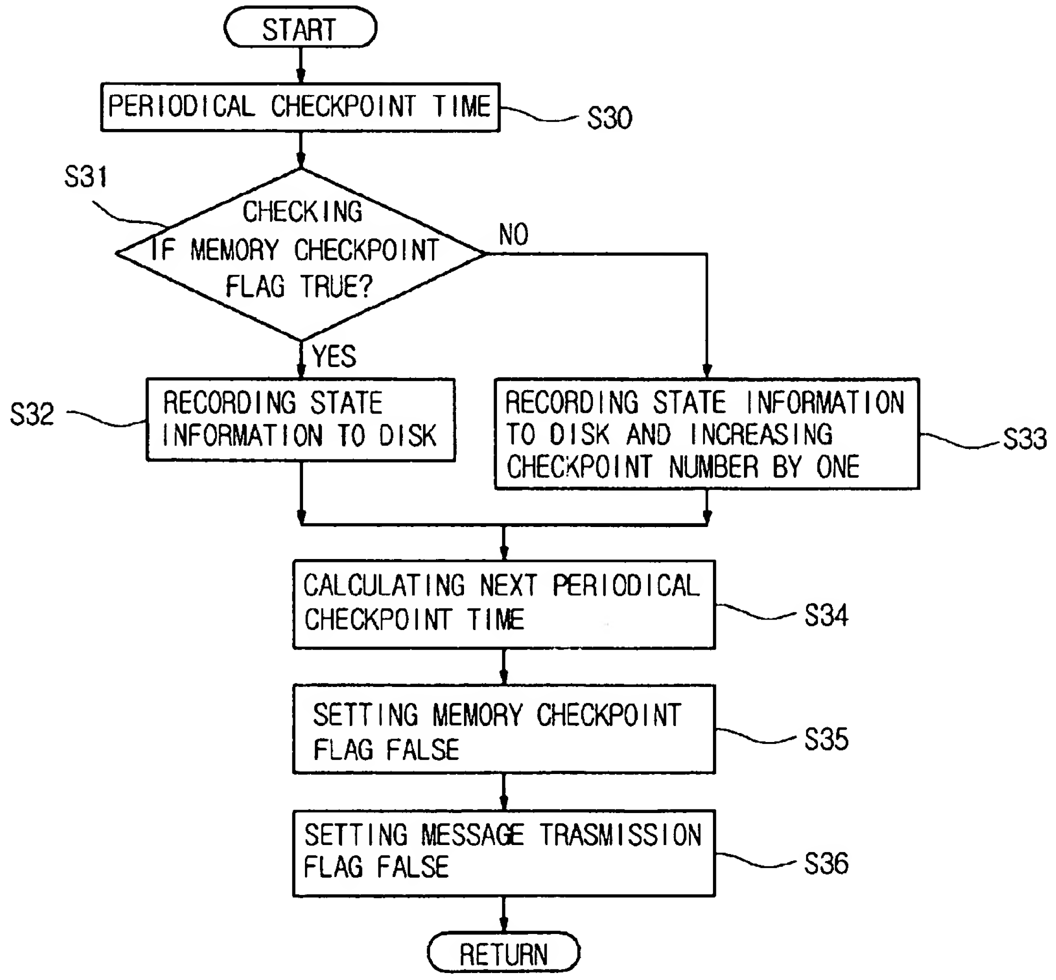

START
RECEIVING A MESSAGE
S21
CHECKING IF THE CHECKPOINT NUMBER OF TRANSMISSION PROCESS IS LARGER THAN THE CHECKPOINT NUMBER OF CURRENT PROCESS?
S22
NO
YES
CHECKING IF THE CHECKED MEMORY CHECKPOINT FLAG IS TRUE?
S23
NO
YES
REPLACING CHECKPOINT NUMBER OF TRANSMISSION PROCESS WITH CHECKPOINT NUMBER OF CURRENT PROCESS
S24
CHECKING IF THE CHECKED MESSAGE TRANSMISSION FLAG IS TRUE?
S26
NO
YES
RECORDING STATE INFORMATION OF CURRENT PROCESS INTO MEMORY AND SETTING CHECKPOINT NUMBER OF TRANSMISSION PROCESS AS CHECKPOINT NUMBER
S27
SETTING MEMORY CHECKPOINT FLAG TRUE, SETTING MESSAGE TRASMISSION FLAG FALSE
S28
PROCESSING MESSAGE
S25
RETURN

FIG. 4


START
PERIODICAL CHECKPOINT TIME
S30
S31
CHECKING IF MEMORY CHECKPOINT FLAG TRUE?
NO
YES
S32
RECORDING STATE INFORMATION TO DISK
RECORDING STATE INFORMATION TO DISK AND INCREASING CHECKPOINT NUMBER BY ONE
S33
CALCULATING NEXT PERIODICAL CHECKPOINT TIME
S34
SETTING MEMORY CHECKPOINT FLAG FALSE
S35
SETTING MESSAGE TRASMISSION FLAG FALSE
S36
RETURN

INDEPENDENT CHECKPOINTING METHOD USING A MEMORY CHECKPOINT ON A DISTRIBUTED SYSTEM

TECHNICAL FIELD

The present invention relates to an independent checkpointing method, which recovers problems in running a distributed system and reduces work time. In the method in accordance with an embodiment of the present invention, transmitting processes send messages with a self checkpoint number, and receiving processes determine whether memory checkpoint is to be performed in reference to the transmitting processes' checkpoint number, current process checkpoint number, memory checkpoint flag, and message transmission flag before processing the received message. In performing periodical checkpointings, the method makes reference to the result of the memory checkpoint for checkpointing.

BACKGROUND OF THE INVENTION

Checkpointing technology stores status information of each process and recovers errors by using the stored information when errors occur in running distributed system environments.

Several studies have been done in the area of checkpointing technology. An adaptive checkpointing algorithm was proposed by Jian Xu et al. at Proceedings of Fifth IEEE Symposium on Parallel and Distributed Processing in 1993. The adaptive checkpointing algorithm performs checkpointing by checking whether zigzag cycles made by an input message exist. If input messages make zigzag cycles, the adaptive checkpointing algorithm is based upon the fact that a domino effect may be caused. Therefore, if input messages make zigzag cycles, the adaptive checkpointing algorithm performs checkpointing to remove zigzag cycles before the processing of messages.

A lazy checkpointing algorithm was proposed by Wang et al. at Technical report CRHC-92-27. In the lazy checkpointing algorithm, message transmission processes attach a self checkpointing number to messages to be transmitted, and the message receiving processes compare the checkpointing number of the transmitted messages with their checkpoint number before processing the transmitted messages. If the checkpoint number of the transmission process is larger, checkpointing is performed before the message is processed. At the moment, the checkpoint numbers of the two processes become identical.

However, when the amount of message transmission and execution speed difference increases, these checkpointing methods may cause a large number of checkpoints and increase job completion time.

The number of checkpoints is directly related with job completion time in the error-free environment and the roll back distance is directly related with job completion time in the environment with errors. Therefore, a large number of checkpoints and an increase in roll back distance causes delayed job completion time.

SUMMARY OF THE INVENTION

An independent checkpointing method using memory checkpoint on a distributed system is provided.

The independent checkpointing method in accordance with an embodiment of the present invention includes a message transmission routine, a message processing routine, and a periodical checkpoint routine. The message transmission routine adds a self checkpoint number to a message to be transmitted when a current process tries to send a message to another process. The message processing routine performs a memory checkpoint and processes a message in reference to a checkpoint number of a transmission process, a checkpoint number of the current process, a memory checkpoint flag, and a message transmission flag when a message is received from a process. The periodical checkpoint routine performs a checkpoint that records a necessary state information for recovery against faults periodically in reference to the memory checkpoint flag.

Preferably, the message transmission routine includes the following steps. A step is to generate a message to be transmitted. Another step is to add the checkpoint number of the current process to the message to be transmitted. A further step is to set the message transmission flag true for preparing cases in which an orphan message occurs. An additional step is to transmit the message.

Preferably, the message processing routine includes the following steps. A first step is to receive the message from the process and compare the checkpoint number of the transmission process with the checkpoint number of the current process. Another step is to process the received message if the checkpoint number of the transmission process is smaller than or equal to the checkpoint number of the current process. A further step is to check the memory checkpoint flag if the checkpoint number of the transmission process is larger than the checkpoint number of the current process. Another step is to replace the checkpoint number of the current process with the checkpoint number of the message transmission process and process the received message if the checked memory checkpoint flag is true. A further step is to check the message transmission flag if the checked memory checkpoint flag is false. An additional step is to record the state information of the current process into a memory, set the checkpoint number of the current process as the checkpoint number of message transmission process, set the memory checkpoint flag true, set the message transmission flag false, and process the received message if the checked message transmission flag is true. Another additional step is to replace the checkpoint number of the current process with the checkpoint number of the transmission process and process the received if the checked message transmission flag is false.

Preferably, the periodical checkpoint routine includes the following steps. An initial step is to check the memory checkpoint flag on a periodical checkpoint time. Another step is to record the state information stored at the memory to a disk if the checked memory checkpoint flag is true or recording the state information to a disk and increasing the checkpoint number by one if the checked memory checkpoint flag is false. A further step is to calculate a next periodical checkpoint time. An additional step is to set the memory checkpoint flag and the message transmission flag false.

BRIEF DESCRIPTION OF THE DRAWINGS

The embodiments of the present invention will be explained with reference to the accompanying drawings, in which:

FIG. 1 is a diagram illustrating a process in a distributed system in accordance with an embodiment of the present invention;

FIG. 2 is a flow diagram illustrating a message transmission routine in accordance with an embodiment of the present invention;

FIG. 3 is a flow diagram illustrating a message processing routine in accordance with an embodiment of the present invention; and

FIG. 4 is a flow diagram illustrating a periodical checkpointing routine in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The method in accordance with an embodiment of the present invention may be applied to a distributed system, which includes a number of local machines connected through LANs (local area network).

FIG. 1 is a diagram illustrating a process in a distributed system in accordance with an embodiment of the present invention. As shown in FIG. 1, a local machine includes local memory and a local disk. A local machine executes a process allocated to a job. Each process includes message transmission routine, message processing routine, and periodical checkpoint routine for recovery against faults. Each process sends information to other processes through messages.

FIG. 2 is a flow diagram illustrating a message transmission routine in accordance with an embodiment of the present invention. As shown in FIG. 2, the independent checkpointing method includes four steps. A first step is to generate a message to be transmitted. A second step is to attach checkpoint number of current process to the message. A third step is to set transmission flag true for processing orphan messages. A fourth step is to transmit the message.

When the current process is running and a new message needs to be transmitted to a different process running in a different local machine, a message that needs to be transmitted earlier is generated at step S10.

Then, a checkpoint number of the current process is added to the generated message to be transmitted at step S11.

Executing a periodical checkpoint routine that is to be described later may increase the checkpoint number of the current process. When a message arrives from other processes, the checkpoint number of the current process may be replaced by the checkpoint number of the transmission process.

After the checkpoint number is added to the message to be transmitted, the message transmission flag is set true for notifying that the current process has sent a message to another process and then the message is transmitted at step S12 and S13.

The message transmission flag is important because it is necessary to reduce the number of the memory checkpoint when orphan messages are generated. Orphan messages are the messages transmitted after the checkpoint of the message transmission processes and received before the checkpoint of the message receiving processes. When errors occur in the message transmission processes, orphan messages bring back the message receiving processes to the point before the checkpoints. Therefore, in order to avoid generating orphan messages, checkpointing is performed after the message transmission processes send a message.

The method in accordance with an embodiment of the present invention employs message transmission flags to reduce the number of memory checkpoints in overall execution of each process and to prevent orphan messages. That is, memory checkpoint is performed only if the checkpoint number of the message transmission process is larger than checkpoint number of the message receiving process and the current message transmission flag of the message receiving process is true.

FIG. 3 is a flow diagram illustrating a message processing routine in accordance with an embodiment of the present invention.

As shown in FIG. 3, the message processing routine includes the following seven steps. A first step is to receive a message from a process and compare a checkpoint number of the message transmission process with a checkpoint number of the current process. A second step is to process the received message if the checkpoint number of the message transmission process is smaller than or equal to the checkpoint number of the current process. A third step is to check a memory checkpoint flag if the checkpoint number of the message transmission process is larger than the checkpoint number of the current process. A fourth step is to replace the checkpoint number of the current process with the checkpoint number of the message transmission process and process the received message if the memory checkpoint flag checked is true. A fifth step is to check the message transmission flag if the memory checkpoint flag checked is false. A sixth step is to record state information of the current process into memory, set the checkpoint number of the current process as a checkpoint number of message transmission process, set the memory checkpoint flag true, set the message transmission flag false, and process the received message if the message transmission flag checked is true. A seventh step is to replace the checkpoint number of the current process with a checkpoint number of the message transmission process and process the received message if the message transmission flag checked is false.

When a message is received from other processes, the message processing routine compares the checkpoint number of the current process with the checkpoint number of the message transmission process that is added to the message before processing the received message at step S21 and S22.

If the checkpoint number of the message transmission process is smaller than or equal to the checkpoint number of the current process, it means that the processing speed of the message transmission process is slower than or equal to the processing speed of the current process. In such cases, since the memory checkpoint of the current process is not necessary, the current process treats the received message without performing the memory checkpoint at step S25.

However, if the checkpoint number of the message transmission process is larger than the checkpoint number of the current process, it means that the processing speed of the message transmission process is faster than the processing speed of the current process. That is, the current process determines whether the memory checkpoint is to be performed before processing the received message.

The memory checkpoint is performed only if the checkpoint number of the message transmission process is larger than the checkpoint number of the current process, the memory checkpoint has not been performed, and the memory checkpoint after transmitting a message to other processes and periodical checkpoint have not been performed. It reduces number of checkpoints by preventing repetition of memory checkpoints and therefore reduces job completion time, even if a number of messages are received from other processes before the periodical checkpoint is performed.

For this purpose, the memory checkpoint flag is checked to see whether the memory checkpoint has been performed before at step S23.

If the flag is true, which means the memory checkpoint was performed after the periodical checkpoint is performed, the memory checkpoint is not to be performed. Instead, the checkpoint number of the current process is replaced by a checkpoint number of the message transmission process and then a next message is processed at steps S24 and S25.

If the flag is false at step S23, the message transmission flag is confirmed at step S26.

The message transmission flag is a flag describing if the current process has sent any messages to other processes. The message transmission flag is set as true by the message transmission routine and as false after the memory checkpoint or periodical checkpoint is performed.

Therefore, if the message transmission flag is true, which means that the result of the step S26 is true, that is, memory checkpoint or periodical checkpoint is not performed yet, the memory checkpoint that stores the state information of the current process into the memory is performed. Checkpoint numbers become identical by replacing checkpoint number by checkpoint number of message transmission process at step S27.

Then, the memory checkpoint flag is set true and the message transmission flag value is set false at steps S28 and S25 before the next message is processed.

FIG. 4 is a flow diagram illustrating a periodical checkpointing routine in accordance with an embodiment of the present invention. As shown in FIG. 4, the periodical checkpointing routine includes the following steps. A first step is to check the memory checkpoint flag on periodical checkpoint time. A second step is to record process state information stored at memory to disk if the memory checkpoint flag checked is true or recording process state information to disk and increasing the checkpoint number by one if the memory checkpoint flag checked is false. A third step is to calculate a next periodical checkpoint time. A fourth step is to set the memory checkpoint flag and the message transmission flag false.

The periodical checkpoint routine checks whether the memory checkpoint flag is true or false before writing current state information of the process with a certain period at steps S31 and S32.

The memory checkpoint flag is a flag indicating the memory checkpoint has been performed in the message processing routine. If the memory checkpoint flag is true, which means the memory checkpoint has been performed recently, the process state information stored in memory is written to the disk without increasing the checkpoint number at step S32.

If the memory checkpoint flag is false, which means the memory checkpoint has not been performed recently, the process state information is written on the disk while increasing the checkpoint number by 1 at step S33.

Then, the amount of time for executing the periodical checkpoint is calculated and the memory checkpoint flag and message transmission flag are set as false at steps S34 through S36.

When the amount of message transmission and execution speed difference increase, conventional checkpointing methods cause a large number of checkpoints, which increases delay caused by checkpoints and eventually causes increased job completion time.

However, the method in accordance with an embodiment of the present invention only performs checkpoint on the memory when a message is initially received from a process with a larger checkpoint number. Later, the method increases only the checkpoint number without performing any checkpoints. As shown in FIG. 4, process state information stored in the memory is recorded to disks in periodical checkpoints.

That is, process state information is recorded to disks with a certain period. Even though memory checkpoint is caused in every period, the time for memory checkpoint is substantially smaller than the time for the disk checkpoint, which means that actual job delay time is trivial.

On the contrary, to the conventional checkpoint methods, the method in accordance with an embodiment of the present invention performs only one disk checkpoint with a certain period.

In addition, as shown in FIG. 3, the method in accordance with an embodiment of the present invention removes dependent relations between processes that are usually caused by message transmissions by comparing checkpoint numbers. That is, when a process receives a message with a larger checkpoint number than the process, the process performs the memory checkpoint before processing the message and identifies a checkpoint number with the transmission process. This procedure virtually removes difference of execution speeds and reduces dependent relations between processes. As a result, the method in accordance with an embodiment of the present invention has a short roll back distance when errors occur.

As described above, the method of the present invention adds a checkpoint number in transmitting messages between processes. When a checkpoint number of message transmission process is larger than a checkpoint number of a current process, the method of the present invention performs the memory checkpoint in reference to a memory checkpoint flag and message transmission flag, replaces the checkpoint number of the current process with a checkpoint number of the message transmission process. In addition, the method of the present invention uses the result of the memory checkpoint in performing a periodical checkpoint.

Therefore, when a number of processes perform the same jobs in a distributed system environment, the method in accordance of an embodiment of the present invention has minimum number of checkpoints regardless of the amount of message transmission and execution speed difference. Also, the method removes dependent relations between processes that are usually caused by message transmissions by comparing checkpoint numbers. The method in accordance with an embodiment of the present invention has a short roll back distance when errors occur. Consequently, the method is effective for completing jobs in a minimum amount of time in a distributed system environment with errors or in a distributed system environment without errors.

Although representative embodiments of the present invention have been disclosed for illustrative purpose, those who are skilled in the art will appreciate that various modifications, additions and substitutions are possible without departing from the scope and spirit of the present invention as defined in the accompanying claims and in equivalents thereof.

What we claim:

1. An independent checkpointing method using a memory checkpoint on a distributed system, comprising:

- a message transmission routine for adding a self checkpoint number to a message to be transmitted when a current process tries to send a message to another process;
- a message processing routine for performing a memory checkpoint and processing a message in reference to a checkpoint number of a transmission process, a checkpoint number of the current process, a memory checkpoint flag, and a message transmission flag when a message is received from a process; and

a periodical checkpoint routine for performing a checkpoint that records a necessary state information for recovery against faults periodically in reference to the memory checkpoint flag.

2. The independent checkpointing method of claim 1, wherein the message transmission routine comprises the steps of:

generating the message to be transmitted;

adding the checkpoint number of the current process to the message;

setting the message transmission flag to true for preparing cases in which an orphan message occurs; and

transmitting the message.

3. The independent checkpointing method of claim 1, wherein the message processing routine comprises the steps of:

receiving the message from the process and comparing the checkpoint number of the transmission process with the checkpoint number of the current process;

processing the received message if the checkpoint number of the transmission process is smaller than or equal to the checkpoint number of the current process;

checking the memory checkpoint flag if the checkpoint number of the transmission process is larger than the checkpoint number of the current process;

replacing the checkpoint number of the current process with the checkpoint number of the message transmission process and processing the received message if the checked memory checkpoint flag is true;

checking the message transmission flag if the checked memory checkpoint flag is false;

recording the state information of the current process into a memory, setting the checkpoint number of the current process as the checkpoint number of message transmission process, setting the memory checkpoint flag true, setting the message transmission flag false, and processing the received message if the checked message transmission flag is true; and

replacing the checkpoint number of the current process with the checkpoint number of the message transmission process and processing the received message if the checked message transmission flag is false.

4. The independent checkpointing method of claim 3, wherein the periodical checkpoint routine comprises the steps of:

checking the memory checkpoint flag on a periodical checkpoint time;

recording the state information stored in the memory to a disk if the checked memory checkpoint flag is true or recording the state information to a disk and increasing the checkpoint number by one if the checked memory checkpoint flag is false;

calculating a next periodical checkpoint time; and

setting the memory checkpoint flag and the message transmission flag false.

* * * * *

US005423068A

United States Patent [19]

Hecker

[11] **Patent Number:** **5,423,068**

[45] **Date of Patent:** **Jun. 6, 1995**

[54] **METHOD OF MANAGING, IN A TELECOMMUNICATION NETWORK, USER DATA OF A USER WHO MAY MOVE FROM A BASE WORK AREA ASSOCIATED WITH A BASE EXCHANGE TO ANOTHER WORK AREA ASSOCIATED WITH ANOTHER EXCHANGE**

[75] Inventor: **Hubertus P. J. Hecker**, Voorburg, Netherlands

[73] Assignee: **Koninklijke PTT Nederland N.V.**, Groningen, Netherlands

[21] Appl. No.: **73,792**

[22] Filed: **Jun. 8, 1993**

[30] **Foreign Application Priority Data**

Jun. 19, 1992 [NL] Netherlands 9201090

[51] **Int. Cl.**⁶ **H04B 7/00**

[52] **U.S. Cl.** **455/56.1**; 455/33.2; 379/60

[58] **Field of Search** 455/33.1, 33.2, 54.1, 455/56.1; 379/59, 60, 63

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,775,999	10/1988	Williams	455/33.2 X
5,101,501	3/1992	Gilhousen et al.	455/56.1 X
5,127,100	6/1992	D'Amico et al.	455/33.1
5,189,734	2/1993	Bailey et al.	455/33.2

OTHER PUBLICATIONS

Rueppel et al, "Feind hört mit", Jan. 1992. pp. 46–49, Technische Rundschau.

Fuhrmann, "Intelligente Vermittlungsfunktionen für Mobilkommunikationsnetze-Teil 2", May 1990, pp. 380-385, NTZ Nachrichten Technische Zeitschrift, No. 5.

Ballard et al, "Cellular Mobile Radio as an Intelligent Network Application", 1989, pp. 389-399, Electrical Communication, No. 4.

ETSI/GSM Technical Specification, GSM 03.03, version 3.5.0, Jan., 1991, European Digital Cellular Telecommunication System (phase 1; Numbering, Addressing and Identification), European Telecommunications Standards Institute, France, 1992.

ETSI/GMS Technical Specification, GSM 03.08, version 3.7.0, Jan., 1991, European Digital Cellular Telecommunication System (phase 1; Organization of Subscriber Data), European Telecommunications Standards Institute, France, 1992.

ETSI/TC GSM,GSM 11.30, Mobile Services Switching Centre, Apr. 9, 1990.

ETSI/GSM, GSM 11.31, Home Location Register Specification, version 3.2.1, Apr. 10, 1990.

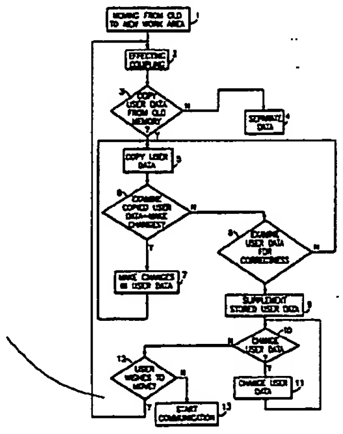
ETSI/GSM, GSM 11.32, Visitor Location Register Specification, version 3.2.1. Apr. 10, 1990.

Primary Examiner—Edward F. Urban
Attorney, Agent, or Firm—Frishauf, Holtz, Goodman & Woodward

[57] **ABSTRACT**

A method of managing, in a telecommunication network, user data of a user who may move from a base work area associated with a base exchange which comprises a base memory to another work area associated with another exchange which comprises another memory, independent of whether or not the user is communicating. The method comprises effecting a coupling between the base memory and the another memory when the user moves from the base work area to the another work area; copying a largest portion of the user data stored in the base memory and storing the copied data in the another memory via the coupling, the user data stored in the base memory remaining available for further copying, and changes to be made in the user data taking place both in the base memory and in the another memory. The copied portion of the user data stored in the another memory is inspected for correctness with respect to the user data stored in the base memory and, in the event of correctness of the copied portion of the user data, the portion of the user data stored in the another memory is made available for further copying. Changes to be made in the data take place only in the another memory, and the copied portion of the user data stored in the base memory thereafter becomes unavailable for further copying.

16 Claims, 1 Drawing Sheet


MOVING FROM OLD TO NEW WORK AREA
1
EFFECTING COUPLING
2
3
COPY USER DATA FROM OLD MEMORY
N
Y
SEPARATE DATA
4
COPY USER DATA
5
6
EXAMINE COPIED USER DATA—MAKE CHANGES?
N
Y
MAKE CHANGES IN USER DATA
7
8
EXAMINE USER DATA FOR CORRECTNESS
N
SUPPLEMENT STORED USER DATA
9
CHANGE USER DATA
10
N
Y
CHANGE USER DATA
11
12
USER WISHES TO MOVE?
N
Y
START COMMUNICATION
13

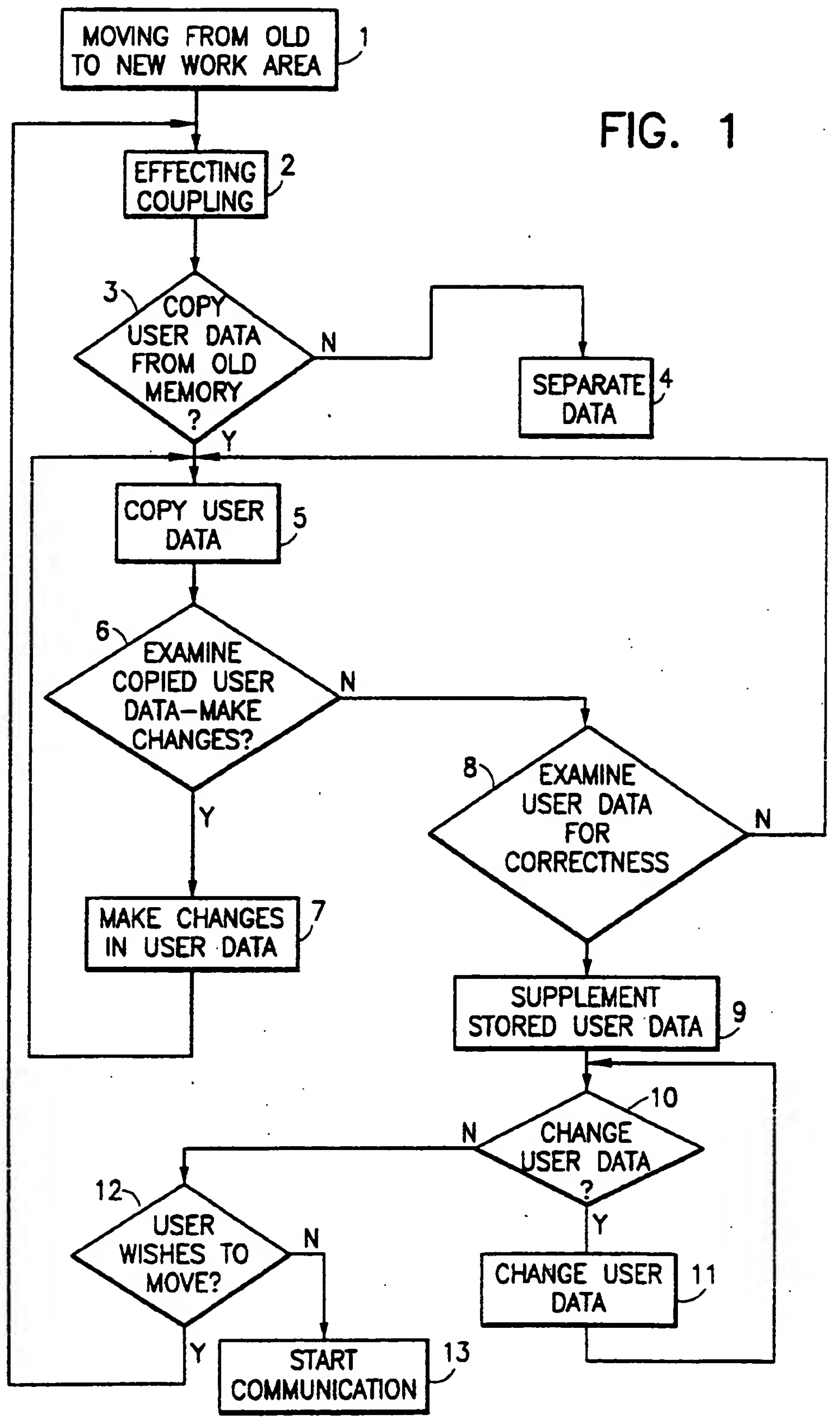

MOVING FROM OLD TO NEW WORK AREA
1
EFFECTING COUPLING
2
3
COPY USER DATA FROM OLD MEMORY ?
N
Y
SEPARATE DATA
4
COPY USER DATA
5
6
EXAMINE COPIED USER DATA–MAKE CHANGES?
N
Y
MAKE CHANGES IN USER DATA
7
8
EXAMINE USER DATA FOR CORRECTNESS
N
SUPPLEMENT STORED USER DATA
9
10
CHANGE USER DATA ?
N
Y
CHANGE USER DATA
11
12
USER WISHES TO MOVE?
N
Y
START COMMUNICATION
13

FIG. 1

METHOD OF MANAGING, IN A TELECOMMUNICATION NETWORK, USER DATA OF A USER WHO MAY MOVE FROM A BASE WORK AREA ASSOCIATED WITH A BASE EXCHANGE TO ANOTHER WORK AREA ASSOCIATED WITH ANOTHER EXCHANGE

BACKGROUND OF THE INVENTION

The invention relates to a method of managing, in a telecommunication network, user data of a user who may move from a base work area associated with a base exchange which comprises base memory means to another work area associated with another exchange which comprises other memory means.

A method of this type is disclosed in various GSM (Group Spéciale Mobile) recommendations, such as GSM 03.03, 03.08, 11.30, 11.31 and 11.32. According to the method described therein, the base memory means associated with the base exchange (Home Location Register or HLR) comprise all the data of the user who, normally speaking, is located in the base work area associated with the base exchange. If the user moves to the other work area associated with the other exchange, only those data which are required for handling calls and for other purposes (which are of importance for the user temporarily located in said other work area) are copied out of the base memory means (HLR) into the other memory means associated with the other exchange (Visitor Location Register or VLR). Under these circumstances, the base memory means (HLR) function as "master" memory means and the other memory means (VLR) function as "slave" memory means, some user data thus being present in two different memories while the user is located in the work area associated with the other exchange. If the user moves to a further work area associated with a further exchange, again only those data which are required for handling calls and for other purposes (which are of importance for the user temporarily located in the said further work area) are copied from the base memory means (HLR) into the further memory means (VLR) associated with the further exchange, the base memory means (HLR) again functioning as "master" memory means and the further memory means (VLR) functioning as "slave" memory means. The base memory means (HLR) therefore always function as "master" memory means, while all the other memory means (VLRs) always function as "slave" memory means, at least as regards this user.

A known method of this type has the disadvantage that some user data are present in two different memories while the user is located in another work area associated with an exchange other than the base exchange, which takes up memory capacity and consequently reduces the efficiency of the method, and that, if the user continues to move to successive work areas, the data required have to be copied over ever greater distances from the base memory means (HLR) to succeeding memory means (VLRs), which reduces both the reliability and the efficiency of the method.

SUMMARY OF THE INVENTION

The object of the invention is, inter alia, to provide a more efficient and more reliable method.

For this purpose, the method according to the invention has the characteristic that, if the user moves from the base work area to the other work area, a coupling is effected between the base memory and the other memory, a largest portion of the user data stored in the base memory being copied and stored via the coupling in the other memory and the user data stored in the base memory being definitive (i.e., being available for copying) for the user and changes to be made in the user data taking place both in the base memory and in the other memory, the portion of the user data copied into the other memory then being inspected for correctness with respect to the user data stored in the base memory, whereafter, in the event of correctness, the portion of the user data stored in the other memory becomes definitive for the user (i.e., becomes available for further copying) and changes to be made in the data take place only in the other memory means, the copied portion of the user data stored in the base memory becoming negligible (i.e., becoming unavailable for further copying).

The method according to the invention rejects the constant functioning of the base memory as "master" memory and the constant functioning of the other memory as "slave" memory means as regards a particular user. Because the largest portion of the user data stored in the base memory is copied and stored in the other memory, the copied portion of the user data stored in the other memory is then inspected for correctness with respect to the user data stored in the base memory, the portion of the user data stored in the other memory becomes definitive (available for copying) for the user after correctness has been confirmed and the copied portion of the user data stored in the base memory thereafter becomes negligible and consequently overwriteable, the memory capacity required (unavailable for copying) for said user data in the base memory is saved, which increases the efficiency of the method according to the invention. If the user then moves to a succeeding work area associated with a succeeding exchange which comprises a succeeding memory, the largest portion of the user data stored in the other memory is copied and stored in the succeeding memory. The copied portion of the user data stored in the succeeding memory is subsequently inspected for correctness with respect to the user data stored in the other memory, the portion of the user data stored in the succeeding memory becomes definitive (available for copying) for the user after correctness has been confirmed and the copied portion of the user data stored in the other memory then becomes negligible(unavailable for copying), as a result of which the required data are no longer copied over the generally large distance between the base memory and the succeeding memory but over the generally smaller distance between the other memory and the succeeding memory which increases both the reliability and the efficiency of the method according to the invention compared with the known method.

The present invention is based on the insight that the storing of the same data in two different memories is inefficient and that the reliability and the efficiency of a method is increased if the data are copied over a smaller distance, which is achieved by storing data in a memory situated as near to the user as possible.

A first embodiment of the method according to the invention has the characteristic that, in the event of incorrectness of the copied portion of the user data stored in the other memory with respect to the user data stored in the base memory, the largest portion of the user data stored in the base memory is copied once more and is stored via the coupling in the other memory.

This increases the reliability of the method further.

A second embodiment of the method according to the invention has the characteristic that user data not belonging to the largest portion and stored in the base memory comprises information relating to a location of the other exchange.

As a result of storing the location of the other exchange in the uncopied portion of the user data in the base memory, where the user is located is permanently known in the base exchange, which promotes the reliability.

A third embodiment of the method according to the invention has the characteristic that the user data not belonging to the largest portion and stored in the base memory comprises information relating to a protection of the user.

As a result of storing information relating to the protection of the user in the uncopied portion of the user data in the base memory, this information is permanently available at one point and it is not copied and transmitted to another memory, which promotes the protection of the user and consequently the reliability.

A fourth embodiment of the method according to the invention has the characteristic that user data not belonging to the copied portion and stored in the other memory comprises information relating to this other exchange.

As a result of storing the information relating to the other exchange in the uncopied portion of the user data in the other memory, this information is available only at that point where the information is necessary and the base memory and succeeding memory are not encumbered therewith, which promotes the efficiency.

A fifth embodiment of the method according to the invention has the characteristic that, in the event of a further movement of the user from a first further work area associated with a first further exchange which comprises a first further memory to a second further work area associated with a second further exchange which comprises second further memory, a further coupling is effected between the first further memory and the second further memory, a further largest portion of the user data stored in the first further memory being copied and stored via the further coupling in the second further memory, and the user data stored in the first further memory means being definitive (available for copying) for the user and changes to be made in the user data taking place both in the first further memory and in the second further memory, the copied portion of the user data stored in the second further memory then being inspected for correctness with respect to the user data stored in the first further memory, whereafter, in the event of correctness, the portion of the user data stored in the second further memory becomes definitive (available for copying) for the user and changes to be made in the data take place only in the second further memory, the copied portion of the user data stored in the further memory becoming negligible (unavailable for copying).

This embodiment illustrates that, if the user has already moved from the base work area to the first further work area and the largest portion of the user data have been copied and stored in the first further memory, and if the user then moves to the second further work area, the largest portion of the user data is in that case copied from the first further memory into the second further memory, which will generally take place over a smaller distance than if copying had to take place from the base memory.

A sixth embodiment of the method according to the invention has the characteristic that, in the event of incorrectness of the copied portion of the user data stored in the second further memory with respect to the user data stored in the first further memory, the further largest portion of the user data stored in the first further memory is copied once again and is stored via the further coupling in the second further memory.

This increases the reliability of the method still further.

A seventh embodiment of the method according to the invention has the characteristic that the information belonging to the user data stored in the base memory and relating to a location of the other exchange is replaced by information relating to a location of the second further exchange.

As a result of storing the location of the second further exchange in the uncopied portion of the user data in the base memory, where the user is located is again permanently known in the base exchange, which again promotes the reliability.

It is pointed out that the movement of a user from the first further work area (such as, for example, the base area) to the second further work area (such as, for example, the other work area) is to be understood either as the movement of the user with his own terminal (such as in the case of mobile radial) or the movement of the user to another terminal (such as in the case of the follow-me feature). The movement of the user results in a movement of the largest portion of his user data to a new location but does not automatically imply that he also actually starts to communicate from his new location.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a flow chart of the method according to the invention.

DETAILED DESCRIPTION

In the flow chart shown in FIG. 1, the blocks have the following meaning:

block	meaning
1	user indicates that he is moving from an old work area to a new work area
2	coupling is effected between old memory means and new memory means
3	user data in the old memory means are examined for the necessity to be copied: no, to block 4 yes, to block 5
4	the portion of the user data present in the old memory means and not to be copied is or has been separated from the portion to be copied and is not copied; this refers, for example, to data relating to an old exchange belonging to the old work area or, if the old work area is a base work area and an old exchange is a base exchange, data relating to a location of the new exchange or relating to a protection of the user
5	the portion to be copied of the user data present in the old memory means is copied and transmitted to the new memory means
6	the copied portion of the user data is examined for changes to be made: yes, to block 7 no, to block 8
7	the changes to be made are made in the copied portion of the user data present in the old memory means; back to block 5 (this has the consequence that the portion to be copied of the user data present in the old memory means is copied once

-continued

block	meaning
	again and is transmitted to the new memory means; of course, it is also possible to make the changes in both memory means, as a result of which renewed copying and transmitting becomes superfluous)
8	the user data stored in the new memory means are examined for correctness: no, back to block 5 yes, to block 9
9	the user data stored in the new memory means become definitive for the user and are supplemented by information relating to the new exchange, the copied portion of the user data present in the old memory means becomes negligible and information belonging to user data present in the base memory means and relating to a location of the old exchange is replaced by information relating to a location of the new exchange
10	the user data stored in the new memory means are examined for changes to be made: yes, to block 11 no, to block 12
11	the changes to be made are made in the user data present in the new memory means; back to block 10
12	whether the user wishes to move again is examined: yes, back to block 2 no, to block 13
13	communication can be started

The method according to the flowchart shown in FIG. 1 proceeds as follows. As soon as a user indicates that he is moving from an old work area (such as his base work area or a first further work area) to a new work area (such as another work area or a second further work area) (block 1), a coupling is effected between old memory means (such as base memory means or first further memory) and new memory means (such as other memory means or second further memory means) (block 2). This indication takes place in the case of mobile radio as a result of the user and his terminal being located between two base stations and the succeeding base station measuring a stronger radio signal than the preceding base station and it takes place in the case of normal telephony as a result of the user switching on a follow-me feature or informing a succeeding telephone set of his presence, for example, by means of a chip card. User data in the old memory means are then examined for the need to be copied (block 3). If the old memory means are the base memory means, the information relating to the location of the new exchange and relating to the protection of the user, for example, will not need to be copied and if the old memory means are the first further memory means, the information relating to the first further exchange associated therewith, for example, will not need to be copied. In this way, the portion of the user data not to be copied in the old memory means is or has been separated from the portion which has indeed to be copied (block 4) and this lastmentioned portion is copied and transmitted via the coupling to the new memory means (block 5).

At this instant, all the user data are still located in the old memory means and the largest portion of said user data, which has been copied, is located in the new memory means. This portion is examined for any changes to be made (block 6). If these exist, they are either made in the user data present in the old memory means, whereafter these modified data are copied and transmitted again (block 5) or they are made both in the old and the new memory means, as a result of which the recopying and retransmission become superfluous (block 7). The user data stored in the new memory means are then examined for correctness (block 8). This can be done by comparing them with the data present in the old memory means or by means of error-detecting and/or error-correcting codes. If correctness is not confirmed, the copied portion of the data stored in the old memory means is recopied and retransmitted (block 5) and if correctness is in fact confirmed, said user data stored in the new memory means become definitive for the user and are supplemented with information relating to said new exchange (block 9). Furthermore, the copied portion of the user data present in the old memory means becomes negligible (unavailable for copying), which implies that they can be overwritten and that the memory capacity used therefor can be reused and the information belonging to the user data present in the base memory means and relating to a location of the old exchange is replaced by information relating to a location of the new exchange.

The user data stored in the new memory means are then examined for changes to be made (block 10). If there are changes to be made, these are only made in the user data stored in the new memory means (block 11) and, furthermore, whether the user wishes to move once again is examined again (block 12). If this is so, a further coupling is made from the new memory means to the yet newer memory means (block 2), etc., and if this is not so, a communication can be started (block 13).

I claim:

1. A method of managing, in a telecommunication network, user data of a user who may move from a base work area associated with a base exchange which comprises a base memory to another work area associated with another exchange which comprises another memory, independent of whether or not the user is communicating, comprising:

effecting a coupling between the base memory and said another memory when the user moves from the base work area to said another work area;

copying a largest portion of the user data stored in the base memory and storing the copied data in said another memory via the coupling, the user data stored in the base memory remaining available for further copying, and changes to be made in the user data taking place both in the base memory and in said another memory;

inspecting the copied portion of the user data stored in said another memory for correctness with respect to the user data stored in the base memory; and

in the event of correctness of the copied portion of the user data, making the portion of the user data stored in said another memory available for further copying and wherein changes to be made in the data take place only in said another memory, the copied portion of the user data stored in the base memory thereafter becoming unavailable for further copying.

2. The method of claim 1, wherein, in the event of incorrectness of the copied portion of the user data stored in said another memory with respect to the user data stored in the base memory, the largest portion of the user data stored in the base memory is copied once again and is stored via the coupling in said another memory.

3. The method of claim 2, wherein user data not belonging to the largest portion and stored in the base memory comprises information relating to a location of said another exchange.

4. The method of claim 3, wherein the user data not belonging to the largest portion and stored in the base memory comprises information relating to a protection of the user.

5. The method of claim 3, wherein user data not belonging to the copied portion and stored in said another memory comprises information relating to said another exchange.

6. The method of claim 3, wherein:

in the event of a further movement of the user from a first further work area associated with a first further exchange which comprises a first further memory to a second further work area associated with a second further exchange which comprise a second further memory, further comprising:

effecting a further coupling means between the first further memory and the second further memory;

copying a further largest portion of the user data stored in the first further memory and storing said copied further largest portion of the user data via the further coupling in the second further memory, the user data stored in the first further memory being available for further copying and changes to be made in the user data taking place both in the first further memory and in the second further memory;

inspecting the copied portion of the user data stored in the second further memory for correctness with respect to the user data stored in the first further memory; and

in the event of correctness of the inspected user data stored in the second further memory, the portion of the user data stored in the second further memory becoming available for further copying and wherein changes to be made in the data take place only in the second further memory, the copied portion of the user data stored in the first further memory becoming available for further copying.

7. The method according to claim 6, wherein, in the event of incorrectness of the copied portion of the user

data stored in the second further memory with respect to the user data stored in the first further memory, the further largest portion of the user data stored in the first further memory is copied once again and stored via the further coupling in the second further memory.

8. The method according to claim 7, wherein the information belonging to the user data stored in the base memory and relating to a location of said another exchange is replaced by the information relating to a location of the second further exchange.

9. The method according to claim 6, wherein the information belonging to the user data stored in the base memory and relating to a location of said another exchange is replaced by the information relating to a location of the second further exchange.

10. The method of claim 2, wherein the user data not belonging to the largest portion and stored in the base memory comprises information relating to a protection of the user.

11. The method of claim 2, wherein user data not belonging to the copied portion and stored in said another memory comprises information relating to said another exchange.

12. The method of claim 1, wherein user data not belonging to the largest portion and stored in the base memory comprises information relating to a location of said another exchange.

13. The method of claim 12, wherein the user data not belonging to the largest portion and stored in the base memory comprises information relating to a protection of the user.

14. The method of claim 12, wherein user data not belonging to the copied portion and stored in said another memory comprises information relating to said another exchange.

15. The method of claim 1, wherein the user data not belonging to the largest portion and stored in the base memory comprises information relating to a protection of the user.

16. The method of claim 1, wherein user data not belonging to the copied portion and stored in said another memory comprises information relating to said another exchange.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,423,068

DATED : June 6, 1995

INVENTOR(S) : Hecker

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Column 2, line 14, delete "means";
lines 31 and 32, delete "(unavailable for copying)".

Col. 4, line 36, after "of" insert --an embodiment of--.

Column 7, line 20 (claim 6), delete "means".

Signed and Sealed this

Fourteenth Day of January, 1997

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

--

US005265229A

United States Patent [19]

Sareen

[11] **Patent Number: 5,265,229**

[45] **Date f Patent: Nov. 23, 1993**

[54] **SINGLE LOAD, MULTIPLE ISSUE QUEUE WITH ERROR RECOVERY CAPABILITY**

[75] Inventor: **Bimal K. Sareen**, Marlborough, Mass.

[73] Assignee: **Digital Equipment Corporation**, Maynard, Mass.

[21] Appl. No.: **547,661**

[22] Filed: **Jul. 2, 1990**

[51] **Int. Cl.**⁵ **G06F 13/00**

[52] **U.S. Cl.** **395/425**; 364/260; 364/260.2; 364/265.1; 364/244.3; 364/DIG. 1

[58] **Field of Search** 395/425, 575

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,451,880	5/1984	Johnson	395/425
4,482,956	11/1984	Tallman	395/250
4,558,429	12/1985	Barlow	395/425
4,779,234	10/1988	Kaneko	365/221
4,807,111	2/1989	Cohen	395/250
4,894,797	1/1990	Walp	395/425
4,949,301	8/1990	Joshi	395/425

FOREIGN PATENT DOCUMENTS

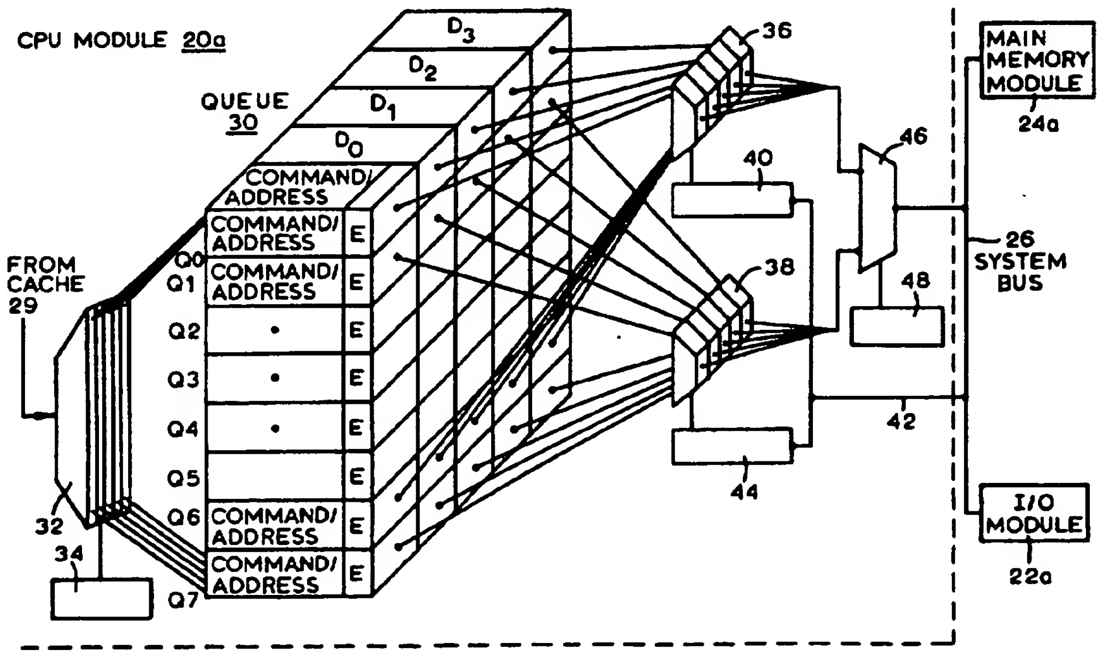
8400836	1/1984	PCT Int'l Appl. .
8504776	10/1985	PCT Int'l Appl. .
8602510	4/1986	PCT Int'l Appl. .
8810468	12/1988	PCT Int'l Appl. .
8907795	8/1989	PCT Int'l Appl. .

Primary Examiner—Eric Coleman
Attorney, Agent, or Firm—William P. Skladony

[57] **ABSTRACT**

An interleaved output queue is used as a high performance interface on a system bus for transferring information from a CPU to main memory. The queue is loaded on its input side with information that is bound for transmission from the CPU's cache to main memory. The queue itself is logically divided into those queue entry addresses which are either odd or even. On its output side, the queue is unloaded by dual sets of unload circuitry, each of which accesses the information stored in either the odd or even queue entry addresses. Other select circuitry will alternate the transmission of information out of the two sets of unload circuitry to main memory. Each set of unload circuitry receives error information back from main memory during the time that the other unload circuitry is issuing a transaction. As a result, each set of unload circuitry is thereby informed whether its own previous transaction was error free, in which case it will transmit the information contained in its next queue entry address when it next issues a transaction, or whether its previous transaction was not error free, in which case it will next have to resend the same information sent during the previous transaction. Consequently, transactions issued from the queue can proceed in an immediately successive sequence without waiting for the processing of error information from the immediately preceding transaction yet the queue can still recover form errors during high performance operation.

27 Claims, 6 Drawing Sheets



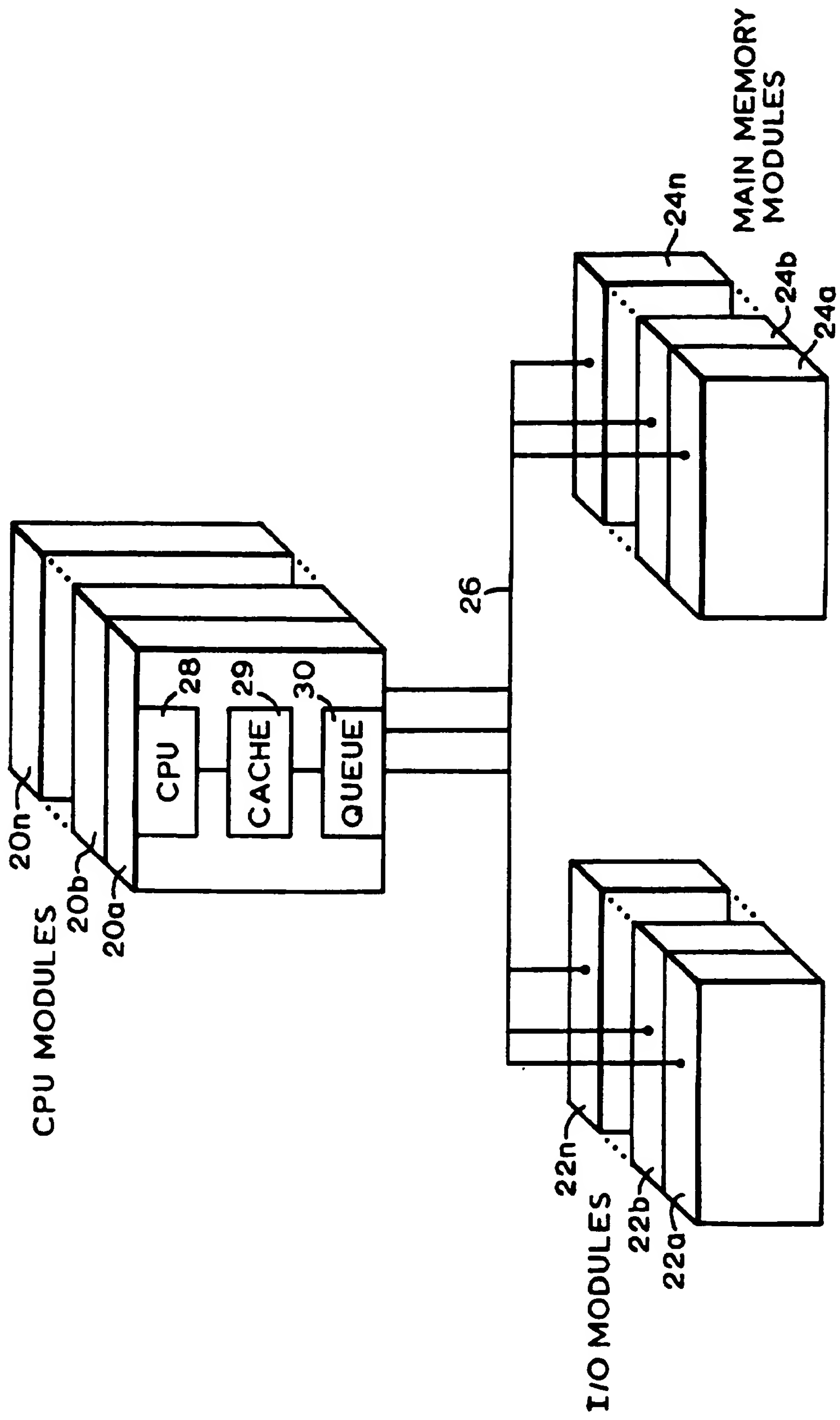

CPU MODULES
20n
20b
20a
CPU
28
CACHE
29
QUEUE
30
26
MAIN MEMORY MODULES
24n
24b
24a
I/O MODULES
22n
22b
22a

FIGURE 1

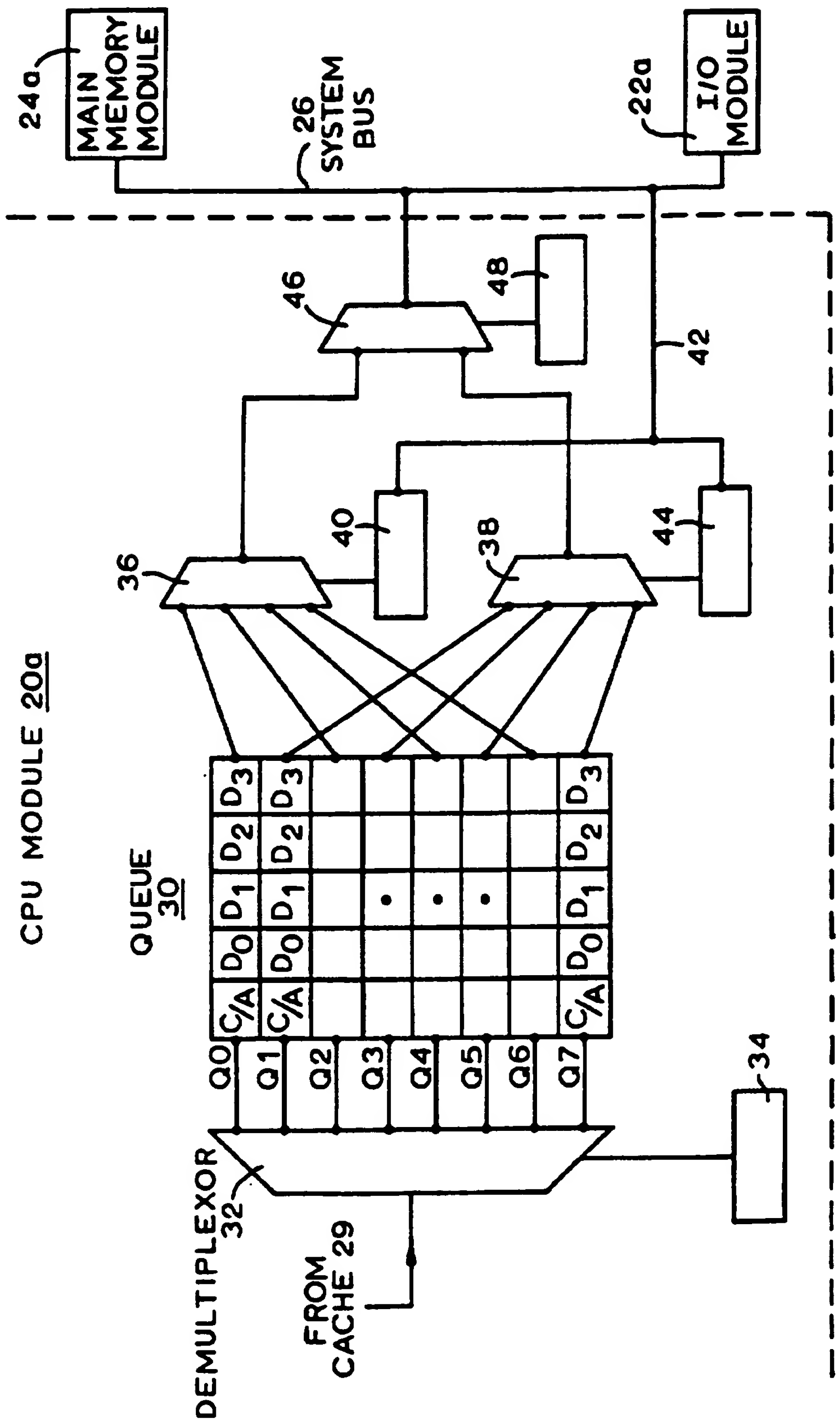

24a
MAIN MEMORY MODULE
26 SYSTEM BUS
22a
I/O MODULE
46
48
42
36
40
38
44
CPU MODULE 20a
QUEUE 30
C/A
D_0
D_1
D_2
D_3
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
34
DEMULTIPLEXOR 32
FROM CACHE 29

FIGURE 2

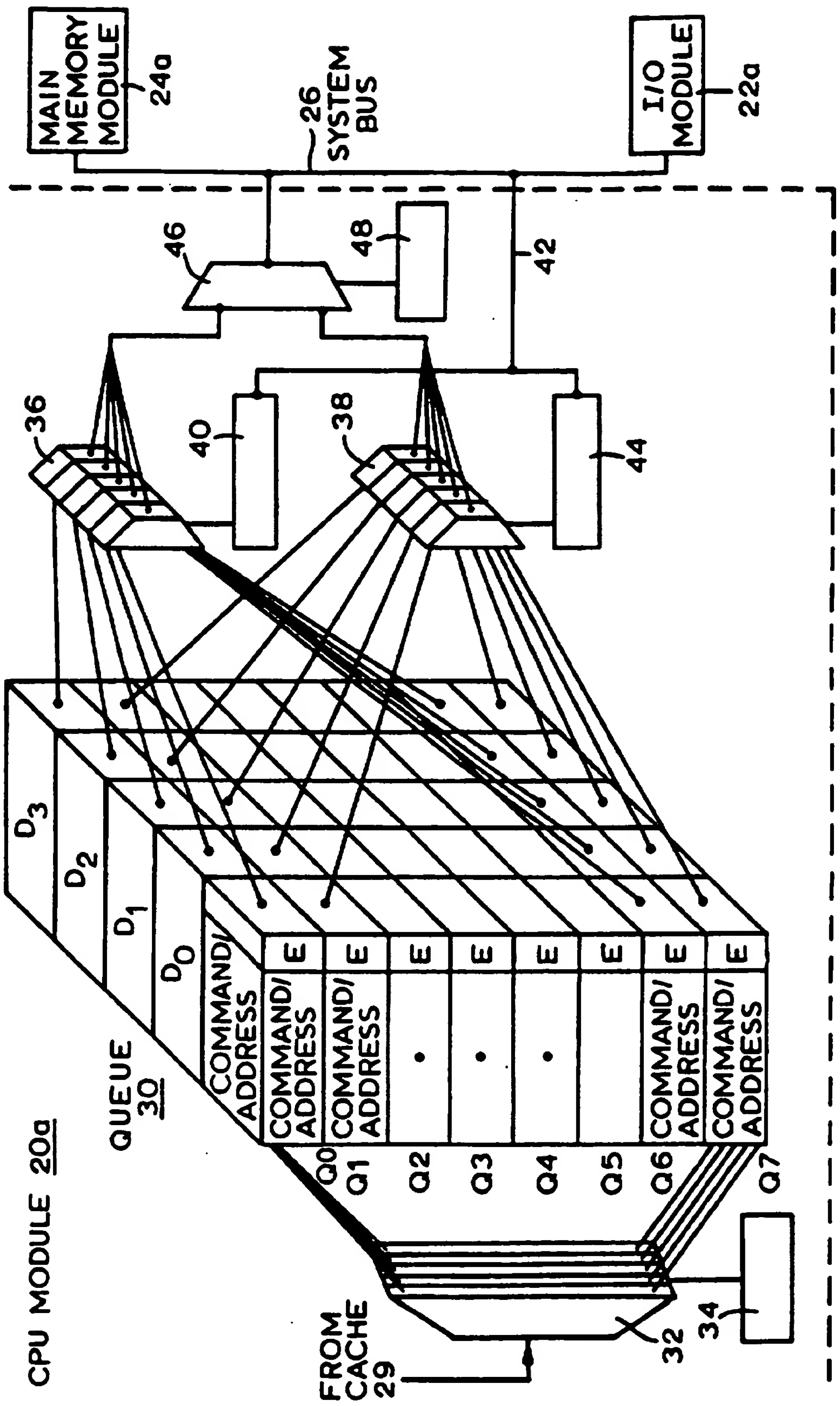

MAIN MEMORY MODULE
24a
26 SYSTEM BUS
I/O MODULE
22a
46
48
42
36
40
38
44
D3
D2
D1
D0
COMMAND/ ADDRESS
E
QUEUE 30
CPU MODULE 20a
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
FROM CACHE 29
32
34

FIGURE 3

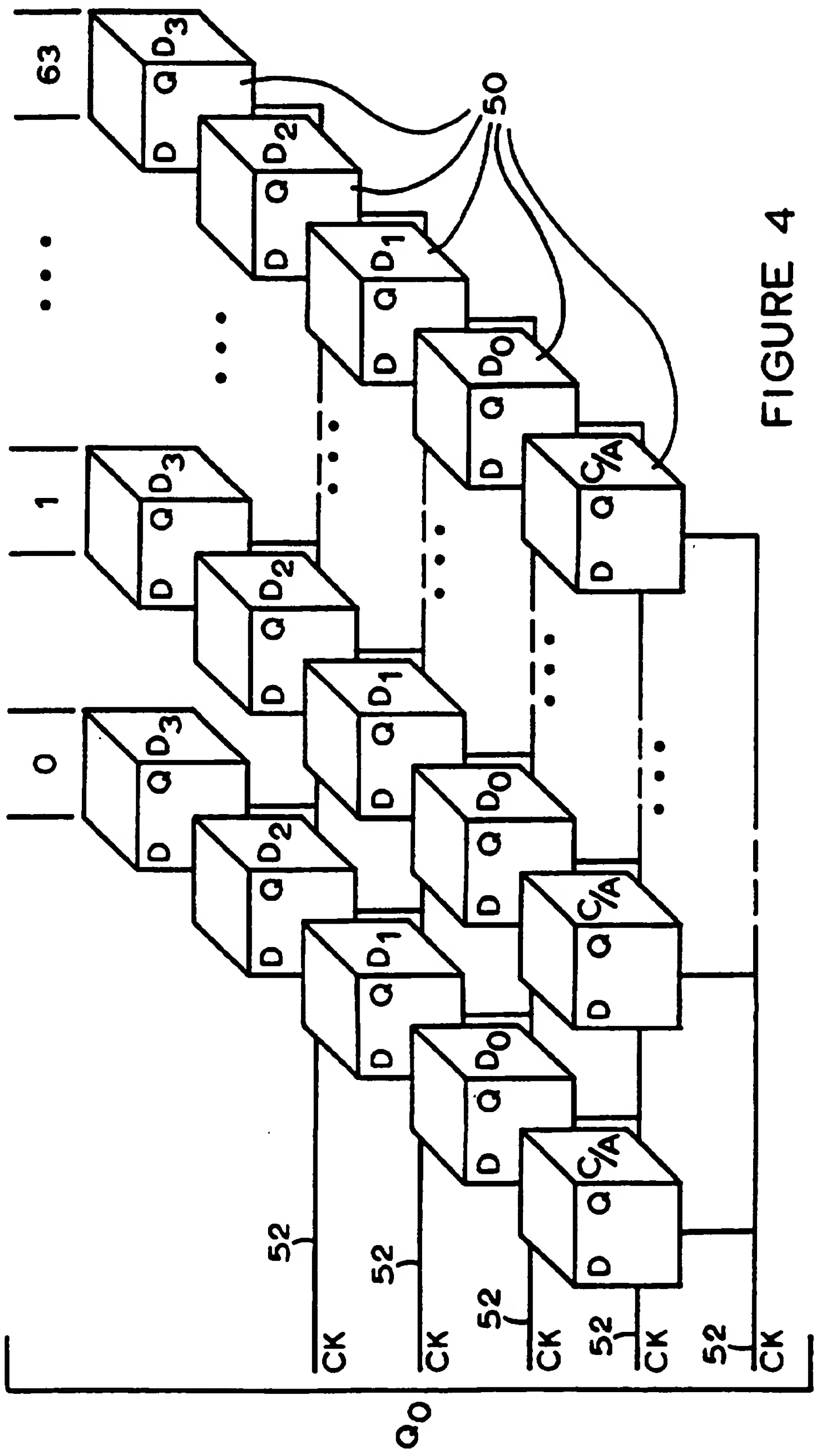
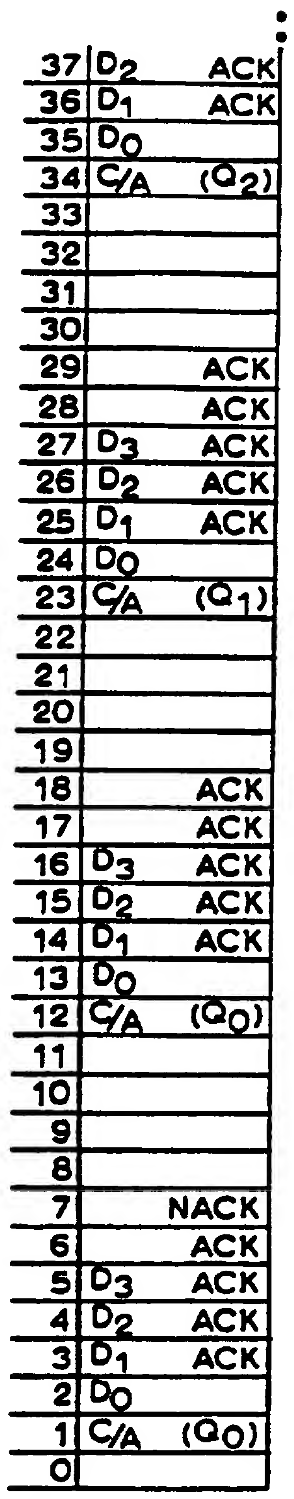


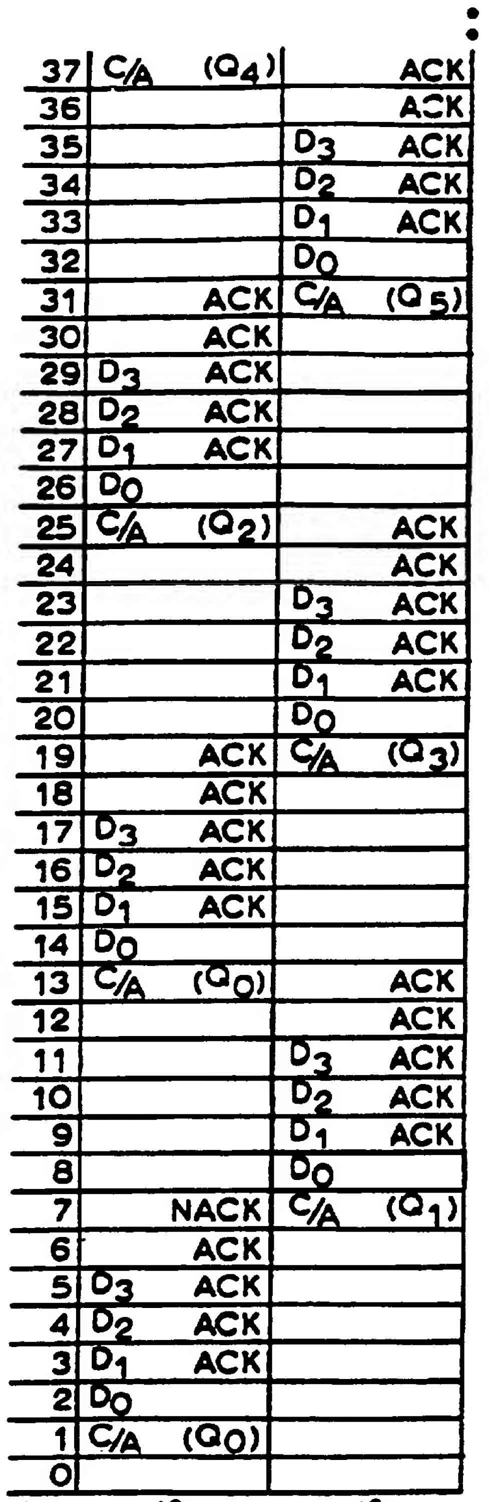
FIGURE 4



QUEUE WITH SINGLE UNLOAD

BUS CYCLE #		
⋮		
37	D_2	ACK
36	D_1	ACK
35	D_0	
34	C/A	(Q_2)
33		
32		
31		
30		
29		ACK
28		ACK
27	D_3	ACK
26	D_2	ACK
25	D_1	ACK
24	D_0	
23	C/A	(Q_1)
22		
21		
20		
19		
18		ACK
17		ACK
16	D_3	ACK
15	D_2	ACK
14	D_1	ACK
13	D_0	
12	C/A	(Q_0)
11		
10		
9		
8		
7		NACK
6		ACK
5	D_3	ACK
4	D_2	ACK
3	D_1	ACK
2	D_0	
1	C/A	(Q_0)
0		

FIGURE 5A



QUEUE WITH DUEL UNLOAD

BUS CYCLE #	EVEN ADDRESS ENTRIES		ODD ADDRESS ENTRIES	
⋮				
37	C/A	(Q_4)		ACK
36				ACK
35			D_3	ACK
34			D_2	ACK
33			D_1	ACK
32			D_0	
31		ACK	C/A	(Q_5)
30		ACK		
29	D_3	ACK		
28	D_2	ACK		
27	D_1	ACK		
26	D_0			
25	C/A	(Q_2)		ACK
24				ACK
23			D_3	ACK
22			D_2	ACK
21			D_1	ACK
20			D_0	
19		ACK	C/A	(Q_3)
18		ACK		
17	D_3	ACK		
16	D_2	ACK		
15	D_1	ACK		
14	D_0			
13	C/A	(Q_0)		ACK
12				ACK
11			D_3	ACK
10			D_2	ACK
9			D_1	ACK
8			D_0	
7		NACK	C/A	(Q_1)
6		ACK		
5	D_3	ACK		
4	D_2	ACK		
3	D_1	ACK		
2	D_0			
1	C/A	(Q_0)		
0				

FIGURE 5B

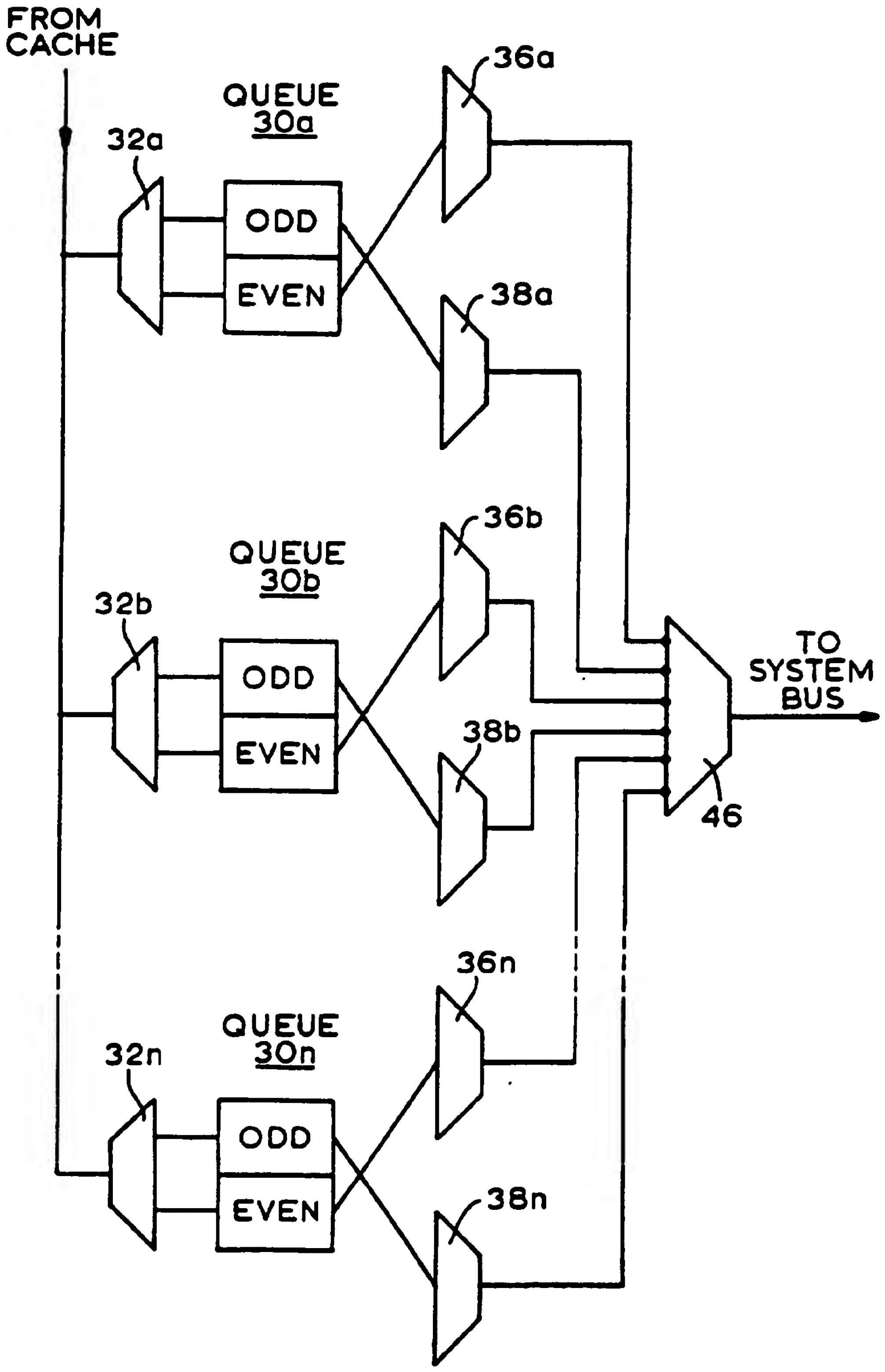

FROM CACHE
QUEUE 30a
32a
ODD
EVEN
36a
38a
QUEUE 30b
32b
ODD
EVEN
36b
38b
QUEUE 30n
32n
ODD
EVEN
36n
38n
46
TO SYSTEM BUS

FIGURE 6

SINGLE LOAD, MULTIPLE ISSUE QUEUE WITH ERROR RECOVERY CAPABILITY

FIELD OF THE INVENTION

The present invention relates to an apparatus and a method for issuing high performance transfers of information from a central processing unit (CPU) within a computer to another location within that computer, while still enabling the CPU to recover from transmission errors on a per transaction basis.

BACKGROUND OF THE INVENTION

Computer systems are typically made up of multiple nodes which are connected together by a system bus which carries information, such as commands, addresses, data, and control signals between the nodes. Examples of the nodes which make up common computer configurations include central processing units, main memory, and I/0 adapters/controllers, which provide interfaces to mass storage devices and networks. When information is transferred over a bus the receiving node will inform the transmitting node whether that information was corrupted during transmission and whether there were any other protocol errors resulting from the transaction. In part, this is accomplished by checking parity bits, which are transferred along with the information.

When the receiving node has confirmed that the transmission is error-free, it will send back an acknowledgment ("ACK") signal to the transmitting node indicating that the information is good. Alternatively, if the information has been corrupted in transmission or any other bus protocol error occurs, the transmitting node does not return an ACK. This failure to return an ACK during the time period required by the bus protocol is logically understood by the transmitting node as a 'no-acknowledgment' ("NACK") signal indicating that the information is not good, and the transmitting node must once again transmit that information.

There is some time delay between the completed transmission of the information to the receiving node, and the determination by the receiving node whether the information was corrupted during transmission and whether there were any other protocol errors. This is because there must be some time for the information to be checked against the parity bit by the bus interface of the receiving node. Thus, if the transmitting node sends information during a first bus cycle, it is common for that node to wait some number of bus cycles before it has an ACK or NACK informing it whether that information is good.

System designs known in the computer architecture art often employ the use of queues for the temporary storage of information which is bound for transmission from one node to another. In a typical implementation, a queue is positioned between a CPU and the interface to the system bus. The CPU will load the queue with information that is to be transferred from the CPU to main memory. An advantage of using the queue in this fashion is that it enhances CPU performance by allowing the CPU to continue processing additional operations before the actual completion of the transfer to main memory. In addition, in the typical implementation the loading of the queue takes place at the same rate at which the CPU operates, which is necessary for maximum performance. The amount of information stored by a output queue is usually sufficient to comprise several transactions. Therefore, a CPU which has a queue is capable of issuing multiple, immediately successive transactions on the system bus as the queue is unloaded.

Although an output queue can issue such multiple, immediately successive transactions, the conventional approach to unloading a queue does not take advantage of this capability because of the need of the system to recover from transmission and bus protocol errors. Specifically, when a NACK results from a given transaction, the system must be able to reissue the transaction so that the information is once again sent from the transmitting node to the receiving node. This conventional approach requires that no successive transaction be commenced until there has been a successful completion of the preceding transaction, or until continuous unsuccessful attempts at transmission of the same information result in a terminating error, known as a "time out".

If, for example, the transmitting node were to commence a second transaction before receiving all of the error information relating to the first transaction, and the first transaction results in an error, the control subsystem of the transmitting node will have to jump back and retransmit the first transaction after transmitting the second transaction. Then, assuming the second transaction is error-free, after the repeated execution of the first transaction, the control sub-system would have to jump ahead to the third transaction, given that there is no need to repeat the second transaction, which was successful. Such a jump back - jump ahead scheme has proven extremely complex, and therefore has not been implemented.

Accordingly, under the conventional unload method, when the information contained in the first queue entry address is transmitted, the system will wait until all of the ACK's have been returned before transmitting the information contained in the second queue entry address. If, however, a NACK is returned, the transaction is repeated. The disadvantage of this approach, however, is that system performance is degraded because the further unloading of the queue is stalled while the node is forced to wait for the ACK's or NACK's relating to a given transaction to be returned. This disadvantage is rather significant in light of the continuing trend for the speed of CPU's to increase, which thereby places a greater demand on system interconnects to issue transactions in an immediately successive order. On the other hand, if the output queue's ability to issue multiple, immediately successive transactions is to be used, the design of the unload system for the queue must be capable of recovering from errors when the original or subsequently attempted transactions result in at least one NACK. Such error recovery capability is a protocol requirement of some advanced system buses.

In accordance with an aspect of the present invention, there is provided a single output queue which is loaded through one set of load circuitry which operates at the same speed as the CPU.

In another aspect of the invention the single output queue is logically divided into two separate logical queues; the two logical queues being made up of those physical queue entry address locations that are even, and those physical queue entry address locations that are odd.

Another aspect of the invention is to access the odd and even queue entry addresses through two separate sets of unload circuitry, which thereby permits rapid unloading of the queue and enhances system performance.

Still another aspect of the invention is to alternate between the unloading of odd and even queue entry addresses so that while one transaction is underway, the immediately preceding transaction can check its own error information to determine whether it should resend the information in the same queue entry address again, or proceed to the next queue entry address. Moreover, this determination by the first half of the logical queue is done independent of the transaction issuance state of the other half of the logical queue. Therefore, the dual access system permits unloading at high performance, but also provides a sophisticated method of error recovery which is practical to implement.

SUMMARY OF THE INVENTION

In accordance with the present invention, an output queue for writing information from a CPU to main memory is logically divided into its odd and even queue entry addresses. The queue has a single load path on its input side, allowing the loading speed to be the same as the operating speed of the CPU. Associated with the queue is a dual set of unload circuitry, one set of which accesses the odd queue entry addresses and the other accesses the even queue entry addresses. Further associated with each set of unload circuitry is an address pointer and error response circuitry. Ultimately, the unload circuitry is connected to the system bus for transmitting information out of the queue to main memory.

After the queue is loaded with information it is emptied through the unload circuitry. Specifically, the even address circuitry accesses the information stored in the first even queue entry address, and transmits that information to main memory. Immediately thereafter, the odd address circuitry accesses the information stored in the first odd queue entry address and transmits that information to main memory. While the second transaction is underway, parity and protocol error information relating to the first transaction is evaluated by the error response circuitry to determine whether the address pointer for the even queue entry addresses should be advanced to the next even address of the queue. If an error in the first transaction is detected, the address pointer will be stalled so that immediately following the completion of the second transaction, the first transaction will be repeated.

Likewise, while the information in the first even queue entry address is being retransmitted, parity and protocol error information relating to the second transaction is evaluated by the error response circuitry to determine whether the address pointer for the odd queue entry addresses should be advanced to the next odd queue entry address. Assuming that the second transaction was error-free, the odd address counter will advance so that the second odd queue entry address information will be transmitted as soon as the retransmission of the first even queue entry address is completed.

The unloading of the queue will generally continue this ordered sequence of accessing odd and even queue entry addresses in like fashion until the queue is empty. Thus, the queue functions as a high performance interface between the CPU and main memory, being loaded on its input side at the same operating speed as the CPU. Moreover, due to its ability to reissue a transaction which resulted in an error, the queue can continue high performance operation even under error recovery conditions.

BRIEF DESCRIPTION OF THE DRAWING

FIG. 1 is a general system overview showing multiple CPU modules each having a write-back queue, multiple I/O modules, and multiple main memory modules all linked together by a system bus.

FIG. 2 is an illustration of the output queue with its related load and unload circuitry.

FIG. 3 is a more detailed illustration of the output queue, using a three dimensional characterization to show the multiple blocks of information stored in the individual queue entry address locations.

FIG. 4 is a representation of the D latches that comprise the output queue storage locations.

FIG. 5A is a timing diagram that shows the performance of an output queue using a conventional, single unload approach, while FIG. 5B is a similar timing diagram that shows the performance of an output queue using the dual unload approach of the present invention.

FIG. 6 illustrates an alternate embodiment of the present invention, showing multiple, dual unload output queues linked together to achieve even higher transaction performance than one, dual unload output queue.

DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention relates to a queue which is loaded through one set of load circuitry which operates at the high performance speed of the CPU, and unloaded through multiple sets of unload circuitry. The exact number of sets of unload circuitry is determined by logically dividing the queue into some number of interleaved, queue entry addresses and issuing transactions from those addresses in an ordered sequence. Consequently, through using this interleaved, transaction issuance approach the queue provides a high performance interface with the system bus.

In the preferred embodiment of the invention the queue is designed to write information from a CPU to main memory through the use of a single, circular queue with dual unload circuitry. A circular queue is a queue which is loaded with information, and as it is unloaded, the queue entry addresses which have been emptied are then available for additional information to be loaded again. The use of this queue substantially improves system performance by having transactions on the system bus proceed one right after the other, without any given transaction having to be acknowledged as error-free before commencing the next transaction. The issuance of transactions in this manner, namely commencing one transaction before receiving back complete error information regarding the immediately preceding transaction, shall hereafter be defined as "multiple, immediately successive transactions". In addition, even though transactions can proceed in this tightly ordered sequence, the interleaved write-back approach is capable of recovering from transmission errors using a sophisticated, but practical to implement, error recovery scheme.

Referring now to the drawing, FIG. 1 shows a computer system, including multiple, identical CPU modules 20(a-n), multiple, identical input/output (I/0) modules 22(a-n), and multiple, identical main memory modules 24(a-n). The illustration shows 'n' number of CPU modules, I/0 modules, and main memory modules because the exact number of CPU, I/O, and main memory modules can vary according to system use and performance objectives. These modules are linked together through a system bus 26, which in the preferred embodiment is Digital Equipment Corporation's XMI bus. Also shown in FIG. 1, the CPU module 20 has a CPU 28 which is connected to a cache 29, which, in turn, is connected to the output queue 30. The queue 30 is positioned between the cache 29 and the system bus 26. Although not shown in FIG. 1, in a multiprocessor environment all of the CPU modules would be identically configured with their own CPU's, caches and queues.

In FIG. 2, the output queue 30 is shown along with the block circuitry that loads it on its input side and unloads it on its output side. Specifically, information, such as commands, addresses, and data, are sent from the cache 29 and are loaded into the queue 30 through the demultiplexor 32, which has an input and an output side and which is controlled by the load control 34. The demultiplexor 32 and its associated load control 34 are responsible for filling the various queue entry addresses of the queue 30 with commands, addresses, and data which are sent by the cache 29 to the main memory module 24. The queue 30 has eight queue entry address locations (Q0-Q7) and each queue entry address location is capable of holding five quadwords (1 quadword=64 bits), one of those quadwords typically being a command (C) together with an address (A) and the other four quadwords being data (D0, D1, D2, D3). Although in this implementation of the invention the queue 30 has eight queue entry addresses and five quadwords in each queue entry address, the actual number of queue entry addresses and words, and the actual word size may vary depending upon design factors and goals.

Although the conventional method of unloading an output queue would involve the use of a single multiplexer to empty the various queue entry address locations, FIG. 2 shows the queue 30 connected to two multiplexers, even queue multiplexer 36 and odd queue multiplexer 38. Through the queue multiplexers 36, 38 the queue 30 can be accessed at multiple points for the purpose of unloading the information contained its eight queue entry address locations. Even though the queue 30 is physically only one queue, it logically has the same functionality as two separate queues; the two logical queues being defined by the queue entry addresses which are either odd or even. The unloading of the queue 30 will proceed with the emptying of first an even queue entry address, such as Q0, then an odd queue entry address, such as Q1, then an even queue entry address, such as Q2, and so on. Furthermore, other implementations of the invention can divide the queue 30 into multiples of any number depending upon design and performance factors; however, additional queue divisions would necessitate the implementation of additional unloading multiplexers, similar to the queue multiplexers 36, 38, and their related unload control circuitry.

The input sides of the queue multiplexers 36, 38 are connected to the output side of the queue 30, and can unload the queue 30 by accessing its eight queue entry address locations. As shown in FIG. 2, even queue multiplexer 36 is connected to the even queue entry addresses of queue 30 (Q0, Q2, Q4, Q6), and odd queue multiplexer 38 is connected to the odd queue entry addresses of queue 30 (Q1, Q3, Q5, Q7). Even queue multiplexer 36 is controlled by even unload control 40 which is a block representation which includes the control circuitry that determines which queue entry address of the queue 30 even queue multiplexer 36 will be accessing during any given transaction. This control circuitry is comprised of an address pointer which enables even queue multiplexer 36 to access each of the even queue entry addresses within queue 30. The incrementing of the address pointer is determined by error response circuitry, which is also contained in even unload control 40.

The error response circuitry receives and reacts to parity and protocol error information, the ACK's and NACK's, resulting from the transmission of information to the main memory 24. After receiving and processing this information the error response circuitry determines whether the address pointer for even queue multiplexer 36 should be advanced to the next address, or should be stalled at the same address because an error has necessitated a retransmission of the information in the same queue entry address location. Therefore, even unload control 40 is connected to the system bus 26 through the error state lines 42. The error state lines 42 carry parity and protocol error information back to even unload control 40 in the form of ACK and NACK signals. The effects that this error information has on accessing information in the queue 30 by queue multiplexers 36, 38 shall be further detailed in connection with the discussion of FIGS. 5A and 5B, below.

As shown in FIG. 2, odd queue multiplexer 38 is similarly controlled by odd unload control 44, the contents and function of which are virtually identical to even unload control 40, except that it determines which odd queue entry addresses of queue 30 odd queue multiplexer 38 will be accessing during any given transaction. Odd unload control 44 contains an address pointer which is controlled by error response circuitry which is similar to the address pointer and error response circuitry of even unload control 40. In addition, odd unload control 44 is connected to the system bus 26 through the error state lines 42 so that the error response circuitry can determine whether the address pointer should be incremented after a given transaction, or whether the same transaction will have to be repeated.

Finally, in FIG. 2 the output sides of the queue multiplexers 36, 38 are connected to the input sides of the select multiplexer 46, while the output side of select multiplexer 46 is connected to the system bus 26. The select multiplexer 46 is controlled by the select control 48, and together they determine which of the two queue multiplexers 36, 38 transmits its information on the system bus 26. Under most circumstances, the select control 48 causes the select multiplexer 46 to toggle between the queue multiplexers 36, 38 so that the select multiplexer 46 alternates between transmitting the information contained in an even queue entry address, then the information contained in an odd queue entry address, then an even queue entry address, and so on. In the meanwhile, the queue multiplexers 36, 38 and their related unload controls 40, 44 are responsible for determining which even or odd queue entry address, respectively, is transmitted to the select multiplexer 46.

FIG. 3 shows the same essential features of the queue 30 and its related loading and unloading circuitry as shown in FIG. 2; however, it does so with enhanced detail. The three dimensional character of the drawing is intended to illustrate that each of the eight queue entry addresses of the queue 30 is actually made up of multiple blocks of information. That information, in the form of commands, addresses, and data, comprises blocks of 64 bits each, and each transaction has associated error status information which is generally designated by the letter "E". The demultiplexor 32 is shown in three dimensional form because it must not only be able to distinguish between the eight queue entry address locations of the queue 30 when it is filling it with information, but it also must be able to distinguish between the five blocks of information at any given queue entry address. Similarly, the queue multiplexers 36, 38 are also shown three dimensionally because they, too, must each be able to distinguish between not only their four respective queue entry addresses when unloading, but also the five blocks of information at any given queue entry address.

All of the actual connections between the demultiplexor 32 and the queue 30, and the queue 30 and the queue multiplexers 36, 38 are not shown in FIG. 3 in the interest of preserving the clarity of the illustration. In an actual implementation of the invention, those connections, however, would be made in a fashion similar to the connections that are shown on FIG. 3, all of those connections being shown generally on FIG. 2.

Referring to FIG. 4, the individual memory cells of the queue 30 are made up of D latches 50 which are enabled using a signal on the clock 52; however, in another embodiment of the invention, the individual memory cells of the queue 30 can be made up of any storage means which meets the performance needs of the system.

FIG. 4 shows the five information blocks, each containing 64 bits, which make up the information stored at one single queue entry address of the queue 30, in this instance Q0 is shown. In the preferred embodiment, the first block contains a command (C), such as a write command, and an address (A), and the next four blocks contain data (D0, D1, D2, D3). The function of the command is to provide main memory with an instruction as to what it is supposed to do with the data which will be following the C/A block, while the address informs main memory what address is to be associated with the command and the data. Thus, a write command would instruct main memory to write the four following blocks of data beginning with the address sent along with the write command.

When a queue entry address is accessed, the given queue multiplexer, 36 or 38, will first send the C/A block stored at that queue entry address as a complete block of 64 bits. The queue 30 will next send D0 as another complete block of 64 bits, followed by D1, and so on until all five blocks at that given queue entry address have been sent. The transmission of these five related blocks of information comprise a single "transaction".

According to the XMI bus protocol, after the transmission of the first block of information, which is made up of the C/A, the main memory 24 must return an ACK signal to confirm that that block of information was without error. If no such acknowledgment is received during the assigned time period, the absence of the confirmation is interpreted as a NACK, which means there has been an transient error in transmission or some protocol error and the information must be retransmitted.

Referring now to FIG. 5A, a timing diagram is shown for an output queue which uses the conventional, single unload method. Along the top of the diagram are bus cycle numbers, below which is shown the information that is transmitted during those bus cycles Assuming that the conventional queue is prepared to transmit information to main memory, during cycle #0, the CPU module will arbitrate for the bus. In this example, access to the system bus has been granted during cycle #0; the granting of access having been determined by the priority level assigned to the CPU module through the arbitration scheme. During cycle #1 the queue will transmit the C/A from the first block of Q0, and during the following four cycles, ##2-5, it will transmit D0-D3 from the remaining four blocks of Q0.

According to the XMI bus protocol, ACK and NACK signals from main memory are not received by the CPU module until two cycles after a given block of information is sent by the CPU module. Therefore, during cycle #3 the queue is sending D1 to main memory over the data lines of the bus, while main memory is returning an ACK, which relates to the C/A, on the separate error state lines. During cycle #4 the queue is sending D2 to main memory, while main memory is returning an ACK which relates to D0, and so on with the succeeding cycles. Accordingly, information and error state signals may be traveling in opposite directions on the bus at any given time because they are on distinct lines.

Still referring to FIG. 5A, the performance limitations of emptying the conventional output queue can be seen. Specifically, after the transmission of the C/A and D0-D3 during cycles ##1-5, which together comprises a single transaction, the complete error state information on that transaction will not be received until cycle #7. During cycles ##8-10 the error logic for the conventional queue will have to process the error information from that first transaction to determine which queue entry address will next be accessed. Given that a NACK was returned during cycle #7, meaning that there was an error resulting from D3, the queue will have to retransmit all of the information contained in Q0. Therefore, during cycle #11 the CPU module arbitrates for and, according to the example, is granted access to the bus. Thereafter, the information contained in Q0 is retransmitted commencing with cycle #12.

Because of the delay in the receipt of error information and because of the need to be able to recover from errors, the conventional approach is to wait until all of the acknowledgment signals have been received before again arbitrating for the bus and attempting the next transaction. As a result, in the conventional method the issuance of the next transaction is simply stalled until the preceding transaction is successfully completed, or a time out error terminates further attempts at transmitting the transaction. On the other hand, it is clear that the overall performance of the unloading system is impaired in that four cycles per transaction are wasted while the system waits for the error logic to process the ACK's/NACK's.

The unloading performance of the queue 30 can be substantially improved through the use of the present invention. Referring now to FIG. 5B, a timing diagram similar to FIG. 5A is shown; however, in this instance, the timing diagram reflects the dual unload approach using the queue 30, the queue multiplexers 36, 38, the unload controls 40, 44, the select multiplexer 46 and the select control 48, as discussed above in connection with FIGS. 2 and 3.

As shown in FIG. 5B, during cycle #0 the CPU module 20 arbitrates for and is granted access to the system bus 26. Similar to the example illustrated through FIG. 5A, the queue 30 likewise commences the unloading of the first even queue entry address, Q0, during cycle #1 by transmitting the first block of information, namely C/A, followed by the four data blocks, D0–D3, during the four succeeding clock cycles. Again, acknowledgments relating to any given block of information will be received by the even unload control 40 two cycles after the information is sent. Consistent with the example given in FIG. 5B, the transmission of D3 has resulted in a NACK, which is received during cycle #7. In this instance, however, during cycle #6 the CPU module 20 has arbitrated for and been granted access to the system bus 26 thus enabling it to commence a second transaction, namely the unloading of the first odd queue entry address, Q1, commencing with cycle #7.

It is important to note that the XMI bus protocol requires at least one bus cycle between transactions during which no command, address, or data information is on the bus, so that other nodes may attempt to gain bus access according to their respective orders of priority. In the example illustrated in this timing diagram, cycle #6 is this inactive cycle, and also in this example no other node has been granted access which delays the continued write-back operation of the queue 30.

Still referring to FIG. 5B, before even unload control 40 receives back all of the error information relating to the first transaction, odd queue multiplexer 38, which accesses the odd queue entry addresses, will transmit the information contained in Q1 commencing with cycle #7. While this second transaction is underway, the error response circuitry contained in even unload control 40 will process the ACK's/NACK's relating to the first transaction, and thereby determine whether the address pointer for even queue multiplexer 36 should be advanced to point to the next even queue entry address of queue 30. If a NACK relating to the first transaction is received, as is shown in FIG. 5B during cycle #7, that address pointer will be stalled at Q0, so that all of the information contained in Q0 is retransmitted, as is shown commencing with cycle #12.

A NACK relating to any one of the five blocks comprising the first transaction would have caused a retransmission of Q0; however, the latest returning error signal, namely the one returned during cycle #7, was chosen to illustrate that the system can recover from errors, with high performance, even if they are not detected until the last return cycle relating to a given transaction.

Similarly, while the information contained in Q0 is retransmitted during cycles ##13–17, the error response circuitry contained in odd unload control 44 will process the ACK's/NACK's relating to the second transaction, and thereby determine whether the address pointer for odd queue multiplexer 38 should be advanced to point to the next odd queue entry address of queue 30. In accordance with the example shown in FIG. 5B, the second transaction involving the transmission of the information contained in Q1 was error-free; all of the ACK's were received. Therefore, the odd address pointer will be incremented so that the fourth transaction will involve the transmission of the next odd queue entry address, Q3, as shown commencing with cycle #19.

Assuming that the second attempt at transmitting the information contained in Q0 is error-free, the even ad-

dress pointer will be similarly incremented during the fourth transaction, which involves the transmission of the information contained in Q3, so that it next points to Q2. After the completion of the transmission of the information in Q3, even queue multiplexer 36 will transmit the information contained in Q2. The alternating transmission of the information contained in the odd and even queue entry addresses of the queue 30 will continue in this ordered sequence until the queue 30 is emptied, or the last pending transaction times out as a result of repeated unsuccessful transmission attempts.

As can be seen by comparing FIGS. 5A and 5B, the present invention can almost double the transmission performance of the queue 30. Referring specifically to the transaction issuance of the single unload queue and the dual unload queue during cycles ##0–13, it can be seen that by the time the single unload queue commences its second transaction, during cycle #12, the dual unload queue is very close to completing its second transaction, that second transaction actually being completed during cycle #13. This very substantial performance improvement is a direct result of the ability of the queue 30 to be able to issue multiple, immediately successive transactions, while maintaining a high performance single load path to the queue 30.

A further advantage of the present invention is that its application may be extended to include multiple output queues. As shown in FIG. 6, 'n' number of identical output queues 30(a–n), with their related demultiplexors 32(a–n), even queue multiplexers 36(a–n) and odd queue multiplexers 38(a–n), are connected to the select multiplexer 46. Although this drawing does not include the detail of FIGS. 2 and 3, each queue would be identical to the queue 30 and its related loading and unloading circuitry as shown and discussed in connection with FIGS. 2 and 3. As can be seen, the present invention can be replicated any number of times if the performance goals of the system require very high speed transaction issuance. Such a design approach could yield significant benefits in a pipelined computer system where high transaction issuance bandwidth is needed.

In addition, the present invention is not limited to the transmission of information between a CPU module and main memory, only. Rather the invention is applicable to transmissions between any high speed cache structures on a pended bus. For example, an I/0 module which is configured with a high speed cache could implement this unload approach to improve its performance.

Accordingly, additional advantages and modifications will readily occur to those skilled in the art. The invention in its broader aspects is therefore not limited to the specific details, representative apparatus, and illustrative examples shown and described above. Thus, departures may be made from such details without departing from the spirit or scope of the invention.

What is claimed is:

1. An apparatus for issuing transactions from a first node within a computer system to a second node within a computer system, said apparatus comprising:

a queue having a plurality of queue entry addresses for electronically storing information;

a loading means for loading information into said queue, said loading means being adapted for electrical coupling with a first node such that with said loading means electrically coupled to a first node said loading means receives information sent from a first node, and said loading means being electrically coupled with said queue such that said loading means loads information into said queue entry addresses;

a first unloading means electrically coupled to a first group of queue entry addresses for transferring information contained in said first group of queue entry addresses to a second node;

a second unloading means electrically coupled to a second group of queue entry addresses, which are different from said first group of queue entry addresses, for transferring information contained in said second group of queue entry addresses to a second node; and

means for alternately enabling said first and second unloading means to issue multiple, immediately successive transactions from said queue, said enabling means being adapted for coupling with a second node for receiving error state information from a second node, which indicates that a given transmission of data from a given queue entry address to a second node was erroneous, said error state information being used by said enabling means to cause a data transmission which results in an error to be repeated.

2. An apparatus as in claim 1, wherein said loading means includes a demultiplexor.

3. An apparatus as in claim 1, wherein said first unloading means includes a queue multiplexer.

4. An apparatus as in claim 3, wherein said enabling means includes an address pointer for indicating which single queue entry address said multiplexor unloads during a given transaction.

5. An apparatus as in claim 4, wherein said enabling means further includes error response circuitry, which is responsive to error state information from a second node for providing a signal to increment said address pointer if the immediately preceding transaction executed by said queue multiplexer did not result in an error.

6. An apparatus as in claim 1, wherein said enabling means includes a select multiplexer which is electrically coupled to said first and second unloading means for determining which one of said first and second unloading means transmits information to a second node during any given transaction.

7. An apparatus for issuing transactions from a central processing unit (CPU) module within a computer system to a main memory, said apparatus comprising:

a queue having a plurality of queue entry addresses for electronically storing information;

a loading means for loading information into said queue, said loading means being adapted for electrical coupling with a CPU module such that with said loading means electrically coupled to a CPU module said loading means receives information sent from a CPU module, and said loading means being electrically coupled with said queue such that said loading means loads information into said queue entry addresses;

a first unloading means electrically coupled to a first group of queue entry addresses for transferring information contained in said first group of queue entry addresses to a main memory;

a second unloading means electrically coupled to a second group of queue entry addresses, which are different from said first group of queue entry addresses for transferring information contained in

said second group of queue entry addresses to a main memory; and

means for alternately enabling said first and second unloading means to issue multiple, immediately successive transactions from said queue, said enabling means being adapted for coupling with a main memory for receiving error state information from a main memory, which indicates that a given transmission of data from a given queue entry address to a main memory was erroneous, said error state information being used by said enabling means to cause a data transmission which results in an error to be repeated.

8. An apparatus as in claim 7, wherein said loading means includes a demultiplexor.

9. An apparatus as in claim 7, wherein said first unloading means includes a queue multiplexer.

10. An apparatus as in claim 9, wherein said enabling means includes an address pointer for indicating which single queue address said queue multiplexor unloads during a given transaction.

11. An apparatus as in claim 10, wherein said enabling means further includes error response circuitry, which is responsive to error state information from a main memory for providing a signal to increment said address pointer if the immediately preceding transaction executed by said queue multiplexer did not result in an error.

12. An apparatus as in claim 7, wherein said enabling means includes a select multiplexer which is electrically coupled to said first and second unloading means for determining which one of said first and second unloading means transmits information to a main memory during any given transaction.

13. An apparatus for issuing write-back transactions from a central processing unit (CPU) module within a computer system to a main memory, said apparatus comprising:

a write-back queue having a plurality of queue entry addresses for electronically storing information, said queue entry addresses being logically divided into even and odd queue entry address locations;

a loading means for loading information into said queue, said loading means being adapted for electrical coupling with a CPU module such that with said loading means electrically coupled to a CPU module said loading means receives information sent from a CPU module, and said loading means being electrically coupled with said queue such that said loading means loads information into said queue entry addresses;

a first unloading means electrically coupled to said even queue entry addresses for transferring information contained in said even queue entry addresses to a main memory;

a second unloading means electrically coupled to said odd queue entry addresses for transferring information contained in said odd queue entry addresses to a main memory; and

means for alternately enabling said first and second unloading means to issue multiple, immediately successive transactions from said queue, said enabling means being adapted for coupling with a main memory for receiving error state information from a main memory, which indicates that a given transmission of data from a given queue entry address was erroneous, said error state information being used by said enabling means to cause a data transmission which results in an error to be repeated.

14. An apparatus as in claim 13, wherein said loading includes a demultiplexor.

15. An apparatus as in claim 13, wherein said first unloading means includes a queue multiplexer.

16. An apparatus as in claim 15, wherein said enabling means includes an address pointer for indicating which single queue entry address said queue multiplexor unloads during a given transaction.

17. An apparatus as in claim 16, wherein said enabling means further includes error response circuitry, which is responsive to error information from a main memory for providing a signal to increment said address pointer if the immediately preceding write-back transaction executed by said queue multiplexer did not result in an error.

18. An apparatus as in claim 13, wherein said enabling means includes a select multiplexer which is electrically coupled to said unloading means for determining which one of said first and second unloading means transmits information to a main memory during any given transaction.

19. A method of unloading information from a first node in a computer to a second node in a computer comprising the steps of:

loading a queue, which has a plurality of queue entry addresses, with information from a first node;

logically grouping all of said queue entry addresses into first and second queue entry address groups, the second group of queue entry address groups being different from the first;

using first and second unloading means, which are each respectively coupled to said first and second queue entry address groups, to unload said queue by having said unloading means alternately transmit information out of their respective queue entry address groups in multiple, immediately successive transactions; and

using error state information from a second node, which indicates that a given transmission of data from a given queue entry address to a second node was erroneous, to cause a data transmission which results in an error to be repeated.

20. The method as in claim 19, further comprising the step of:

using an address pointer for indicating the single queue entry address of a given address group which is unloaded during a given data transmission.

21. The method as in claim 20, further comprising the step of:

incrementing the address pointer to the next address if the error state information from the previous data transmission out of the address group with which the pointer is associated did not indicate that the previous transmission was in error.

22. A computer system comprising:

first and second nodes,

a queue having a plurality of queue entry addresses for electronically storing information which is transferred by said first node to said second node;

a loading means for loading information into said queue, said loading means being electrically coupled with said first node for receiving information sent from said first node, and said loading means being electrically coupled with said queue for loading information into said queue entry addresses;

a first unloading means electrically coupled to a first group of queue entry addresses for transferring information contained in said first group of queue entry addresses to said second node;

a second unloading means electrically coupled to a second group of queue entry addresses, which are different from said first group of queue entry addresses, said second unloading means being for transferring information contained in said second group of queue entry addresses to said second node;

said second node transmitting error state information to said first node indicating whether the transfer of any of information from said first node to said second node was in error; and

means for alternately enabling said first and second unloading means to issue multiple, immediately successive transactions from the queue entry addresses to which they are respectively coupled, said enabling means being coupled to said second node to receive the error state information, which is used by said enabling means to cause a transmission which results in an error to be repeated.

23. An apparatus as in claim 22, wherein said loading means includes a demultiplexor.

24. An apparatus as in claim 22, wherein said first unloading means includes a queue multiplexer.

25. An apparatus as in claim 24, wherein said enabling means includes an address pointer for indicating which single queue entry address said multiplexor unloads during a given transaction.

26. An apparatus as in claim 25, wherein said enabling means further includes error response circuitry which is responsive to error state information from said second node for providing a signal to increment said address pointer if the immediately preceding transaction executed by said queue multiplexer did not result in an error.

27. An apparatus as in claim 26, wherein said enabling means includes a select multiplexer which is electrically coupled to said unloading means for determining which one of said first and second unloading means transmits information to said second node during any given transaction.

* * * * *

US006477564B1

(12) United States Patent
Freyssinet et al.

(10) Patent No.: US 6,477,564 B1
(45) Date of Patent: Nov. 5, 2002

(54) **PROCESS FOR TRANSFORMING AND ROUTING DATA BETWEEN AGENT SERVERS PRESENT IN SOME MACHINES AND A CENTRAL AGENT SERVER PRESENT IN ANOTHER MACHINE**

(75) Inventors: **Andre Freyssinet**, Claix (FR); **Marc Herrmann**, Saint-Etienne de Crossey (FR); **Serge Lacourte**, Saint-Martin d'Heres (FR)

(73) Assignees: **Bull S.A.**, Louveciennes (FR); **Inria**, Le Chesnay (FR)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/332,999**

(22) Filed: **Jun. 15, 1999**

(30) **Foreign Application Priority Data**

Jun. 18, 1998 (FR) 98 07718

(51) **Int. Cl.**^{7} **G06F 15/16**
(52) **U.S. Cl.** **709/202**; 709/102; 709/314
(58) **Field of Search** 709/102, 202, 709/217, 219, 314, 318, 238

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,706,516 A * 1/1998 Chang et al. 709/314
6,052,363 A * 4/2000 Koch 370/252
6,108,782 A * 8/2000 Fletcher et al. 713/153
6,292,827 B1 * 9/2001 Raz 709/217
6,327,630 B1 * 12/2001 Carroll et al. 709/314

FOREIGN PATENT DOCUMENTS

EP	0723236	7/1996
EP	0817042	1/1998
WO	9510805	4/1995

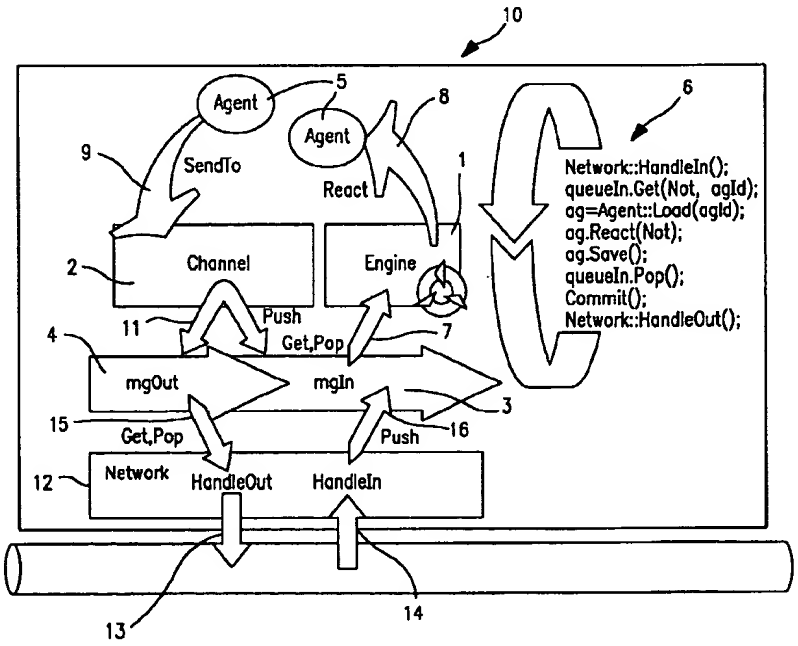
* cited by examiner

Primary Examiner—Ayaz Sheikh
Assistant Examiner—Philip B. Tran
(74) *Attorney, Agent, or Firm*—Miles & Stockbridge PC; Edward J. Kondracki

(57) **ABSTRACT**

The present invention relates to a process for transforming and routing data between agent servers present in some machines and a central agent server present in another machine. The agent server comprises autonomous agents (5) that communicate via notifications, a storage layer and an agent machine comprising an execution engine, a communication channel and two message queues for the notifications, a local queue (mqIn) and an external queue (mqOut). The execution engine takes a notification in the local queue, determines and loads the corresponding agent for each notification, has the agent react to the notification, which agent can then change its status and/or send notifications to the communication channel (2), which stores them in the local queue if they are addressed to a local agent and in the external queue if they are addressed to an agent of another server.

14 Claims, 3 Drawing Sheets



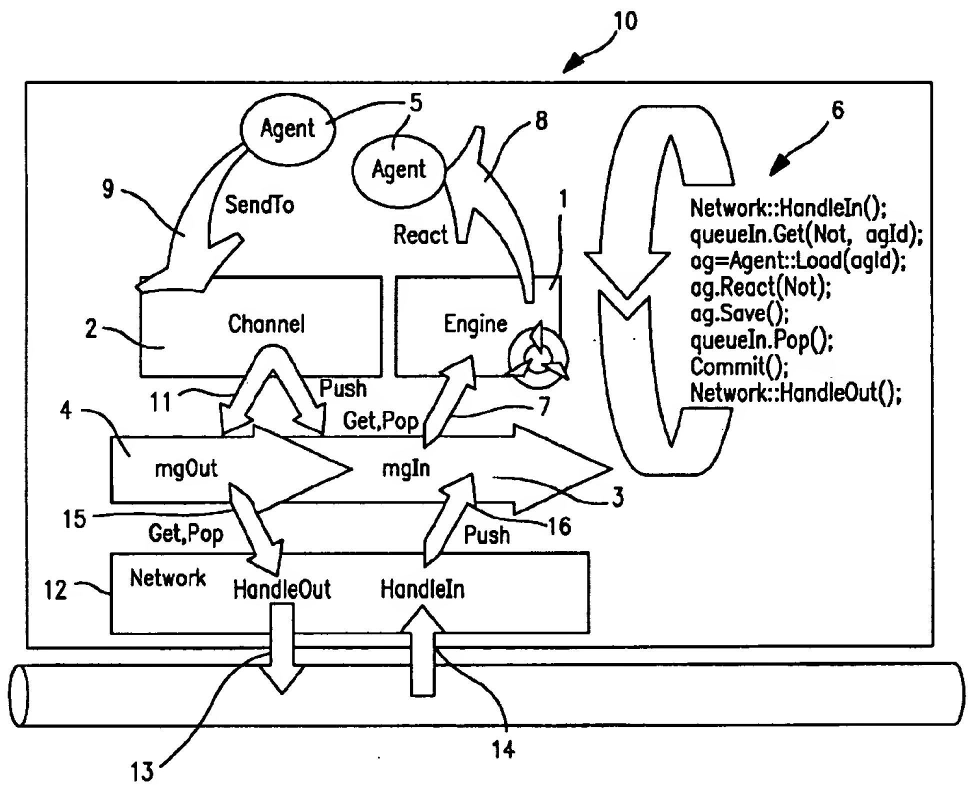

10
5
Agent
Agent
8
6
9
SendTo
React
1
Network::HandleIn();
queueIn.Get(Not, agId);
ag=Agent::Load(agId);
ag.React(Not);
ag.Save();
queueIn.Pop();
Commit();
Network::HandleOut();
2
Channel
Engine
11
Push
Get,Pop
7
4
mgOut
mgIn
3
15
Get,Pop
Push
16
12
Network
HandleOut
HandleIn
13
14

FIG. 1

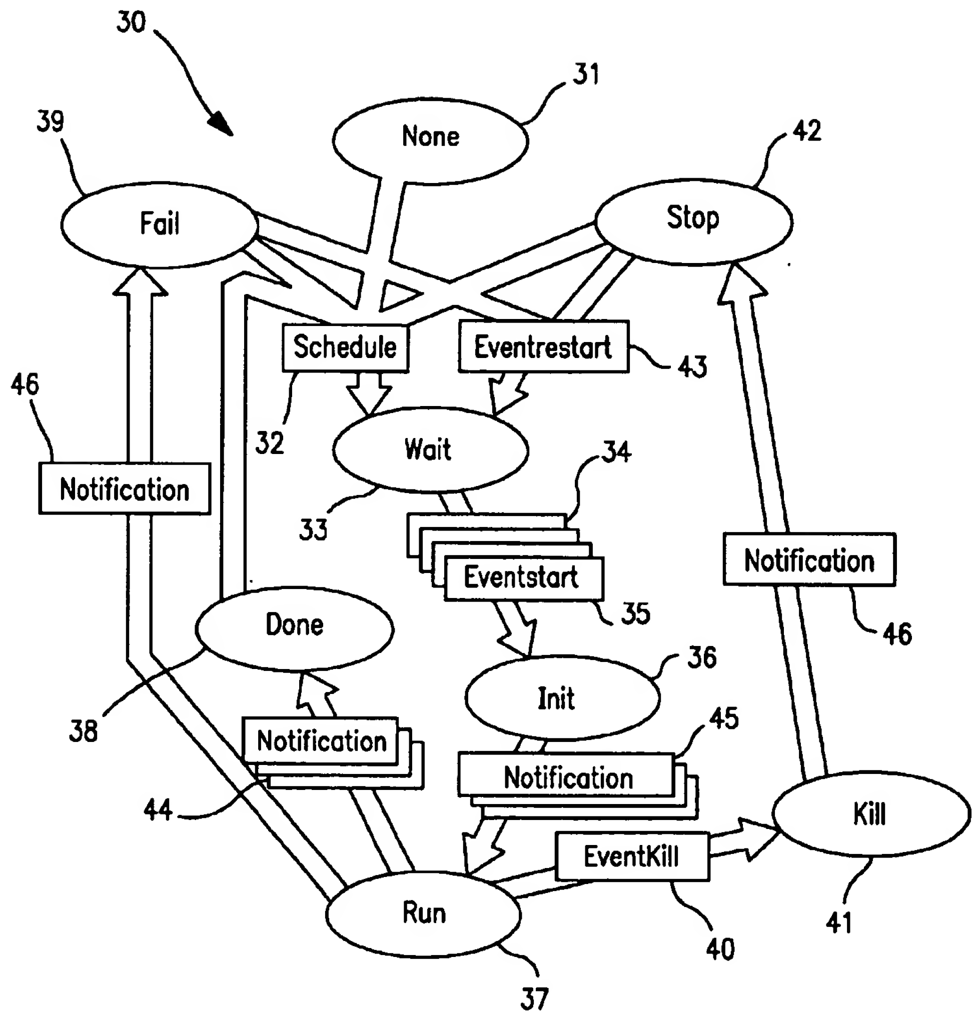

30
None
31
39
Fail
Stop
42
Schedule
Eventrestart
43
32
Wait
46
Notification
33
34
Eventstart
35
Notification
46
Done
36
Init
38
45
Notification
Notification
44
Kill
EventKill
Run
40
41
37

FIG. 2

20

```
class Agent {
 React(Notification) throw Exc;
 Save() throw Exc;
 Load(AgentId) throw Exc;
};
```

FIG. 3

21

```
class Channel {
 SendTo(AgentId, Notification) throw Exc;
};
```

FIG. 4

22

```
Class messageQueue {
 Push(AgentId, Notification) throw Exc;
 Get(AgentId, Notification) throw Exc;
 Pop() throw Exc;
};
```

FIG. 5

PROCESS FOR TRANSFORMING AND ROUTING DATA BETWEEN AGENT SERVERS PRESENT IN SOME MACHINES AND A CENTRAL AGENT SERVER PRESENT IN ANOTHER MACHINE

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates to a process for transforming and routing data between agent servers present in some machines and a central agent server present in another machine.

More specifically, the invention relates to a process for transforming and routing data between agent servers present in machines of production sites and a central agent server present in the machine of the decision-making base, for example within the framework of a centralized concentration of data, also known as a Data Warehouse.

2. Description of Related Art

Data Warehousing is a computing concept aimed at concentrating in a centralized unit, accessible through a network, all of the information available in a company. Data Warehousing allows a better distribution of the information by no longer presenting the information in the form of raw data, but so that it is directly usable by the requestor or client (the company's executives, for example). A Data Warehouse therefore corresponds to one or more of the company's production sites with databases, plus a decision-making base that can retrieve the information in a new form (matching information that can be represented in different ways, transforming or deleting information, etc.). At each site, there is a process for transforming and routing the data for filling this decision-making base. The hardware and software used can be different from one site to another. In the prior art, the machines at each site require a particular software program for interpreting and transforming the information from the database or databases of the site. A software program must therefore be rewritten for each site. The processes that currently exist use the distribution to the various machines of a synchronous process known as a Remote Procedure Call (RPC). The various elementary operations (the extraction, transfer and loading stages) are mixed in one algorithm.

SUMMARY OF THE INVENTION

The object of the invention is to restructure the data transforming and routing software in order to make it easily extendable and to make it open.

This object is achieved through the fact that the process for transforming and routing data between agent servers present in some machines and a central agent server present in another machine is characterized in that an agent server comprises autonomous agents that communicate via notifications, a storage layer (Storage) and an agent machine comprising an execution engine (Engine), a communication channel (Channel) and two message queues for the notifications, a local queue (mqIn) and an external queue (mqOut), and in that the execution engine takes a notification from the local queue (mqIn), determines and loads the corresponding agent for each notification, has the agent react to the notification, which agent can then change its status and/or send notifications to the communication channel, which stores them in the local queue (MqIn) if they are addressed to a local agent and in the external queue (MqOut) if they are addressed to an agent of another server.

According to one characteristic, the execution engine (Engine) of the agent machine executes, after the agent's reaction, a save (Save) of the agent's status, a deletion (Pop) of the processed notification from the local message queue (MqIn) and, atomically, a validation (Commit), of the various operations performed that have changed either the status of the agent or the status of the message queues.

According to another characteristic, each message includes an optimized matrix clock that can be read by the execution engine of the agent machine, thus making it possible to process the notifications in their causal sending order.

According to another characteristic, a message is an object that contains four pieces of information, i.e., the identifier of the sending agent, the identifier of the destination agent, a clock and the notification.

According to another characteristic, the agent machine controls the distribution to all the entities of the network, the reliability of the operations and the scheduling of the messages, which is established by incrementing a logical matrix clock.

According to another characteristic, the message channel performs the following separate transactions:

- a notification sent by an agent is stored locally in a message queue, i.e., either the local queue (mqIn) for the notifications addressed to a local agent, or the external queue (mqOut) for an agent located in another server;
- at the end of the agent's reaction, the two queues are saved on the disk, together with the reacting agent;
- a function (handleout) of the network extracts each notification, in turn, from the external queue (mqOut) and sends it to the target agent server;
- a function (handlein) of the network receives the notifications from the other servers and inserts them into the local queue (mqIn) in causal order.

According to another characteristic, a target agent server receives a message, stores it in its persistent storage layer (Storage), validates the operation (Commit) and confirms the reception of the message to the source server; upon reception of this confirmation, the source server deletes the message from its external queue (mqOut).

According to another characteristic, a persistent external service ensures the atomic nature of the operations that have changed either the status of the agent or the status of the message queues (3,4).

According to another characteristic, the execution engine:

- takes a notification that is in the queue, using the Get function,
- sees which agent the notification is addressed to,
- loads the agent corresponding to the notification (Load),
- has the agent react by calling a reaction operation (React); the agent can change its status by modifying its image in memory, and can send notifications by addressing the channel that will distribute the notification, in which case the notifications are routed to a message queue where they are stored in chronological order,
- if the reaction operation is executed, saves (Save) the status of the agent, which has been changed by the reaction it has executed, deletes (Pop) the notification that has been processed, and atomically validates all of the operations that have been performed (Commit),
- starts again in the program loop in order to process another notification.

According to another characteristic, the data transforming and routing process is executed between agent servers present in machines of production sites and a central server present in the machine of a decision-making base, and uses a particular agent (Event) describing the specific events, in which a status automation is implemented.

According to another characteristic, the status controller comprises various statuses:

a None status in which the event is not active,

a notification (Schedule) causes the event to pass into the Wait status in which the event waits for its activation condition to become true; the event delivers itself a notification that changes it to the Init status;

a simple event attains the Run status directly and ail executes its action; a composed event signals to its sub-events its agreement for them to start and waits for the first of them to attain the Run status in order to pass into the Run status itself;

a simple event then passes into the Done or Fail status, depending on the status of its action; for a composed event, this change depends on the execution statuses of all the sub-events;

a event in execution can be killed by sending it a notification (Eventkill), in which case it passes into the Kill status, marking the fact that it has been killed, and then to the Stop status when it has actually been killed;

a terminated event, i.e., in the Done, Fail or Stop status, can be re-executed by the client by sending it a notification (Schedule), in which case it is reset and passes into the Wait status; the event then propagates the notification to all the sub-events; an event executed repeatedly and endowed with a Done status is automatically reset by the agent Scheduler, which sends it a notification (Schedule);

an event in the Fail or Stop status can be re-executed without being reset by sending a notification (Eventrestart); when this condition becomes "true," the event restarts at the point at which it was previously stopped.

According to another characteristic, a new function can be added to the architecture by adding an agent class derived from the class "Event."

According to another characteristic, three categories of "Event" type agents—"SeqEvent," "ParEvent," "ComplexEvent"—type take on other elementary operations of the "Event" type in order to respectively execute them in sequence, in parallel or in accordance with the dependencies between the various operations.

BRIEF DESCRIPTION OF THE DRAWING

By Other characteristics and advantages of the present invention will emerge more clearly through the reading of the following description provided in reference to the appended drawings, in which:

FIG. 1 represents the architecture of an agent server at a site and the algorithm of the execution engine of the agent machine;

FIG. 2 represents the status controller of an agent "Event";

FIGS. 3, 4 and 5 respectively represent the class "Agent, "the class "Channel," and the class "messageQueue."

DESCRIPTION OF THE DRAWINGS

Schematically, at each production site, there are one or more machines constituted by an agent server. The communications between the agent servers are established through a network, for example a component Network. In each agent server, an agent machine is defined, which in turn has the capability to define agents, with a certain agent model. This agent machine manages a certain number of properties, which include the atomic aspect of an agent's reaction to a notification and the delivery of notifications in a causal sending order. The agent machine comprises various entities. In reference to FIG. 1, the agent machine comprises an execution engine (1) (Engine), a communication channel or message bus (2) (Channel) associated with the execution engine (1), the channel (2) having two queues, a local queue (3) (mqIn) and an external queue (4) (MqOut). The agent server (10) also includes a storage layer (Storage). The queues (3, 4), the channel (2) and the agents (5) are supported by the storage layer.

All the agents (5), the communication channel (2) and the execution engine (1) of the agent machine are C++ objects in a UNIX process. Persistence is obtained by using a DBMS (Data Base Management System). Most of the objects, and not just the agents, can be stored in the persistent storage layer. Each object has a unique, persistent identifier. The objects are compatible with a generic interface in order to be used by common, persistent service functions.

The agents (5) are elementary programming and execution entities. The agents (5) are autonomous in order to give the structure the property of extendibility by making it easy to add and remove agents within the architecture. They are made autonomous by correctly defining the interface of these agents. They are also autonomous in terms of communication. They communicate using notifications and not procedure calls, which makes it possible to open up the architecture. An event is represented by a notification. The notifications pass through the channel (2) and are controlled by the execution engine (1) (Engine) of the agent machine. The notifications are delivered to the agents via the communication channel (2) (Channel). The agents (5) are persistent objects, their status is determined by data that exists outside any execution structure. An agent (5) is an independent machine that reacts to the notifications. All the agents are independent and only the reacting agents need to be loaded into the memory. A "reacting agent" is an agent that has just received a notification, and whose behavior is imposed by the "React" function of the agent called by the execution engine (1) (Engine). These reacting agents can then be replaced on the disk at the end of the program loop of the execution engine (1), which is described below. Each agent is an object of a class and is identified by an unique identifier (Agent Id) which makes it possible to locate it. An agent can be created in a remote server. The creation entity is responsible for creating the identifier. The structure of the identifier must take into account the static location of the agents in order to allow the system to keep track of the notifications and to allow local "production" of identifiers for the remote agents. The identifier of an agent comprises three parts, i.e., the identification of the agent server hosting the creation agent, the identification of the agent server hosting the agent created, and a stamp, local to the agent server hosting the creation agent.

The persistent objects are divided into four classes. All of the objects having the same properties or the same policy for allocating identifiers belong to one class. A defined class can be reused to define a sub-class by adding specific properties to it. The sub-classes inherit properties from the superclass. An inheritance graph defines the hierarchy of the classes. Four basic classes are also defined, the class "globalObject," the class "localObject," the class "notification," and the class "agent."

The objects of the class "globalObject" represent the configuration data defined by the client at the time of the configuration and shared by the various events. Two main categories are described, the hosts in the class "Host" and the derived classes, and the databases in the class "DbConfig" and the derived classes. An object of the class "DbConfig" always points to an object of the class "Host."

The class "localObject" has been defined so as to provide flexibility in the management of persistence. The interface for persistence imposes the movements of all of the objects from the storage layer (storage) to the memory and from the memory to the storage layer, even if some of the objects have not changed during the reaction. Declaring a member of an object in this class makes it possible to explicitly manage the persistence of this member, independently of the persistence of the object itself.

The objects of the class "notification" represent the messages exchanged by the agents. Most of the notifications relate to the management of "warehouse" events.

The agent class represents the "warehouse" events. The object class "Agent" defines the generic interface for all the agents. The object class "Agent" defines the common behavior of all the agents. Each agent is an object of one of the classes derived from the agent class. The agent class provides a mechanism ("swap-in/swap-out") that makes it possible to load the agents or to find them in main storage in order to activate them and to dump the agents not used since a given moment. This is performed by a class "AgentHashTable," which maintains a list of the agents already in memory with the date of their last access. This class is created using a table (Hashtable) with the agent identifier in the first position. The agents can be explicitly deleted from the memory by a "Delete" function which deletes them from the table (Hashtable). In reference to FIG. 3, the object class "Agent" (20) defines two functions, the "Save" function and the "Load" function, which saves and loads the status of the objects into and out of auxiliary storage. The agents, identified by their identifiers, are loaded automatically by the execution engine (1). The reactive behavior is implemented by a "React" function, which defines the reaction of an agent when it receives a notification. This function is called by the execution engine (1) of the agent machine.

A first type of "Agenda" or "scheduler" agent (Scheduler) manages the chronological execution of "warehouse" queries. This agent is activated by the reception of a notification (Schedule) (32). This agent reacts to this notification by selecting the agents it must reactivate by sending an agreement notification (AgreementScheduler). It manages the time and the alarms, and the agents can request to be "reactivated" at a certain time. This agent is present only in the central agent server of the decision-making base.

A second type of agent "factory" (factory) is present in all the agent servers and is responsible for creating objects in the server. It makes it possible to manage the creation of agents remotely.

A third and a fourth type of agent "UnixEvent" and "SigChild" make it possible to manage UNIX processes. "Sigchild" exists in a UNIX agent server and captures the termination signals of "child" processes.

A fifth type of agent "Clientproxy" makes it possible to transcribe commands ordered by a client into notifications. It exists only in the central agent server and supports the relationship with the client.

An agent class "Event" describes the "warehouse" events. A "warehouse" query is composed of several elementary

tasks called By "warehouse" events. In this agent architecture, each event is executed and controlled by a separate agent. All the "warehouse" events have a behavior an properties in common, which results in a more precise definition of a type of agent specific to the Data Warehouse: the agent Event. It must be compatible with a certain interface and a mode of operation. Like any agent, it is a status machine, but in this case the statuses are defined. An object "Event" differs from an object "Agent" by the following attributes:

- an event has a name and a description;
- all the events that form a "warehouse" query are organized hierarchically. The query is identified by an identifier of the event at the root of the hierarchy, and all of the events retain the identifier of their parents and their query;
- the purpose of an Event is to accomplish an action. The start of the execution of the action is conditioned by the reception of a series of agreement notifications (Agreement) (34);
- an object Event uses lists (tables) and produces lists resulting from an extraction, transfer, or transformation operation, for example. The tables produced are established with the notification of the termination of the event's action (ActionDone) (44), and are represented by C++ objects in the class localObject;
- an object Event is a status machine; it maintains a status variable and reports changes in its status to its parents by means of notifications (StatusNotification) (45).

Two main groups of events can be isolated. A basic class (DelegatingEvent) is defined for all the events that perform an elementary operation specific to Data Warehousing: extraction, transfer, transformation, etc. These various operations inherit from the agent Event all the properties of this object. A second basic class of service (ComposedEvent) helps to structure the execution of a query: a sequential, parallel, or more complex execution. This class defines a common behavior for all the composed events such as, for example, the management of the list of sub-events; create, save, restore, delete.

The last important event is the UnixEvent, which includes the execution of a Unix command.

The status changes possible during the execution of the event are represent in FIG. 2. The status graph or status controller (30) is implemented in the agent EVENT in C++:

- the None status (3) is the initial status in which the event is not active;
- a notification (Schedule) (32) causes the event to pass into the Wait status (33) in which the event waits for its activation condition to be come true (True). This condition depends on a series of agreement notifications (Agreement) (34). When this condition becomes true, i.e., when all the agreements have been received, the event delivers itself a notification (Eventstart) (35) in order to start;
- upon reception of this notification (Eventstart) (35), the status is set to Init (36);
- a simple event attains the Run status (37) directly and executes its action. A composed event signals to its sub-events its agreement for them to start, and waits for the first of them to achieve the Run status (37) (it waits for the action to have actually started);
- a simple event passes into the Done (38) or Fail (39) status, depending on the status of its action, upon reception of a notification that the event delivers to

itself (respectively 44, 46). For a composed event, this change depends on the execution statuses of all the sub-events;

an event in execution can be killed by sending it a notification (Eventkill) (40), in which case it passes into the Kill status (41), marking that it has been killed, and then to the Stop status (42) when it has actually been killed, upon reception of a notification (46);

a terminated event, i.e., in the Done (38), Fail (39) or Stop (42) status, can be re-executed by the client by sending it a notification (Schedule) (32).

an even in the Fail (39) or Stop (42) status can be re-executed without being reset by sending a notification (Eventrestart) (43). When this condition again becomes "true," the event restarts at the point at which it was previously stopped.

In the class "Event," a function (Start) makes it possible to change the status and to automatically send the status change to the operation that manages the operation. The "Start" function makes it possible, simply by calling it, to send the necessary notifications automatically and to change the status. The class has a basic implementation for reactions to a certain number of notifications. In particular, there is a procedure in the basic class Event for changing from Wait (33) to Init (36). An agent that inherits from Event, i.e., that is part of a sub-class of the class "Event," retains this management. In the standard mode, the status changes to Init (36), then to Run (37), then to Done (38); from Done (38), the status can return to Wait (33). This is the standard schema. The basic implementation is Wait (33), Init (36), Run (37). In more specific classes, it is necessary to control the change from Init (36) to Run (37) manually. There is a basic implementation for reacting to a stop request signal (40). Upon reception of this Kill notification (41), it will change to the Kill status (41), then to the Stop status (42). This implementation can, for example, be loaded into a sub-class; it is possible in the sub-class to provide another function that will implement this change, possibly separating it into two stages, from Run (37) to Kill (41), then from Kill (41) to Stop (42). There is therefore a basic implementation for all these status changes. When wishing to define a particular class, it is necessary to override the class Event and to provide a implementation with a certain number of functions that follow its rules, and each function, for example the Start function, must necessarily call the change to the Run status (37). In a class, there are variables, each of which contains one of the statuses, and status change functions, each of which triggers the change, for example from Wait (33) to Run (37) or from Run (37) to Done (38). In the class Event, a standard layout is provided, which makes it possible to change directly from Init (36) to Run (37), but a class that is going override it must provide the implementation for the Start function by placing in its code the change from Init (36), followed by the change to Run (37). Thus, introducing a new agent class amounts to inheriting, then providing implementation for new functions, in other words, adding a new function to the architecture amounts to adding an agent class derived from the class Event.

The data transfer process is broken down into elementary tasks which pass through the statuses described above; thus, the execution progresses in elementary steps. Each elementary operation must satisfy the status graph. When, for example, a service is requested of an elementary operation, the operation will pass into Wait (33) and Init (36), and the agent machine (which is described below) guarantees that the operation attains, at a given moment, the Done status (38) or the Fail status (39), or possibly, if a stop request (30) has been requested, the Stop status (42). The organization and synchronization of the elementary tasks are part of a library. Three particular agent classes, "SeqEvent," "ParEvent," and "ComplexEvent," which inherit from the class "ComposedEvent," have been created. An object "SeqEvent" executes its sub-events in sequential order. It reacts to a notification "EventStart" by sending its agreement to the first of its sub-events. When the latter is finished, it passes to the Done status (38). If one of the sub-events fails, the object "SeqEvent" stops and passes into the Fail status (39).

In the same way, an object "ParEvent" executes its sub-events in parallel. It reacts to the notification Eventstart by sending its agreement to all the sub-events. It waits until they are finished and changes its status as a function of their ending statuses. An object "Complex Event" makes it possible for arbitrary dependencies to be set between sub-events. Since each event or elementary operation, or each "Event," must comply with the status graph, it is possible to construct an event that will manipulate sub-events of an unknown type, but which are known to comply with this graph, so that they are executed sequentially. It is possible to start two operations in parallel, for example two extractions that do not involve the same databases and the same sites, and to re-synchronize at the end of these two operations in order to start another operation, for example a transformation. Adding parallel management, for example, while maintaining the reliability of the agent machine (which is described below) is possible as a result of the open and extendable architecture of the process. Thus, different services can be added to the machine simply by adding a class.

The agent machine ensures the distribution of the notifications, the transmission of notifications, and the management of the atomicity of all the operations. The agent machine comprises various entities. In reference to FIG. 1, the agent machine comprises an execution engine (1) (Engine), a communication channel (2) (Channel) associated with the execution engine (1), the channel (2) having two queues, a local queue (3) (mqIn) and an external queue (4) (MqOut). There are well-defined entities and the program (6) of the execution engine (1) uses these various entities.

The interface of the communication channel (2) (Channel) is provided by the class "Channel." According to FIG. 4, the class "Channel" (21) defines a function (Sendto), which sends a notification to an identified agent. The object "Channel" is responsible for the target agent. The object "Channel" provides the following properties: a notification accepted by the channel (2) (Channel) is sent at some point, and two notifications with identical source and target agents are sent in order of their emission.

According to FIG. 5, the class "MessageQueue" (22) defines an addition function (Push) used by the object "Channel" to add a notification to a queue, and two functions (Get and Pop) used by the execution engine (1).

The execution engine (1) is provided by the class "Engine." The class "Engine" is the main part of the architecture. This class "Engine" provides a multiprogramming of the agents. The distributed architecture involves several agent servers running on different machines. Each agent is attached to only one agent server, the migrating entities being the notifications. One of the essential properties of the execution machine in an agent server is the transactional execution of an agent reacting to a notification: either all of the actions, i.e., the notifications sent and the status changes of the agents, are executed, or none of the operations are executed. Each object of the class "Engine" is associated with an object of the class "MessageQueue" defined when the object "Engine" is created. The execution engine (1) (Engine) provides a program loop (6) which successively takes the notifications from the local message queue (3), and calls and executes the appropriate reaction function (React) of the target agent.

In reference to the algorithm (6), of the execution engine (1) (Engine) represented in FIG. 1,

the execution engine (1) takes a notification that is in the local queue (4) using the Get function (7);

the execution engine (1) sees which agent the notification is addressed to;

the execution engine (1) loads the agent that corresponds to the notification (Load);

the execution engine (1) has the agent react by calling a reaction operation (React) (8). The agent can change its status, i.e., modify its image in memory, and can send (9) notifications, i.e., address the channel (2), which will distribute (11) the notification. The notifications are then routed to a message queue (3, 4) or they are stored in chronological order;

if everything runs correctly, i.e., if the reaction operation is executed completely, the execution engine executes a save (Save) of the agent's status, which has been changed by the reaction it has executed, a deletion (pop) of the processed notification from the local message queue (3), and a validation of all the operations that have been performed (Commit). All of these operations have changed either the status of the memory, or the status of the message queues (3, 4) and the matrix clock, which is defined below;

then the execution engine, (1) starts again in the loop in order to process another notification.

This process is distributed among all of the agent servers of the machines. The component Network (12) is responsible for establishing communication between the agent servers. When the agent server is initialized, the potential agents created beforehand are all on the disk in persistent space. When the execution engine (1) sees that it must make a particular agent react because there has been a notification to this agent, it will store it in the memory of the process, possibly from the image saved on the disk. This is the "Load" operation, a movement from the disk to the memory.

Part of the source agent's reaction is to deliver new notifications to the other, so-called target agents. The sending of the notifications is part of the transaction of the reacting source agent server. The possible use of the notification is part of the transaction of the reacting target agent server. The transmission of the notification from the source agent server to the target agent server is not part of the agent's reaction, but is achieved via the channel (2), which executes the following separate transactions:

a notification sent by an agent through the channel (2) is stored locally in a message queue, i.e., either the local queue (3) (MqIn) for the notifications addressed to a local agent, or the external queue (4) (mqOut) for an agent located in another server;

at the end of the agent's reaction, the two queues (3, 4) are saved on the disk, together with the reacting agent. The changes in the agent and in the message queues (3, 4) take place in the same transaction, and are established at the same time;

a function "handleout" of the network Network extracts (15) each notification, in turn, from the external queue (4) (MqOut) and sends it (13) to the target agent server;

a function "handlein" of the network Network receives (14) the notifications from the other servers and inserts

them (15) into the local queue (3) (mqIn) in causal order. When the target agent server receives the message, it stores it in its persistent storage layer (Storage), validates the operation (Commit) and confirms the reception of the message to the source server. Upon reception of this confirmation, the source server deletes the message from its external queue (4) (MqOut). The message is then guaranteed to have been stored persistently by the agent server.

The operations for saving the agent's status, for deleting the processed notification (Save, Pop) and for validating the sent notifications must be intimately linked. The atomicity results from these three operations. It is essential, for example, not to save the agent and leave the notification in the local queue (3), so that the notification is not processed if there is a failure between the two operations. This part must be managed atomically. In order to ensure atomicity between these two operations, the process relies on a persistent external service, which makes it possible to save the agent, the local queue and the external queue simultaneously. Everything is kept in the same persistent service that provides this atomic operation. When changes are made, the persistent service saves the old version of the files, and when the validation operation "commit" is called in this persistent service, it is the persistent service that ensures that the operation is executed atomically. The atomicity includes the modifications in the agent, but also the messages it has sent to the outside and the message it has used. Three things are executed atomically.

If a problem occurs during the reaction, the changes in the database are cancelled (Rollback). Depending on the type of error, it may be necessary to perform additional operations in order to re-synchronize the database and the objects of the memory, and to allow the program to continue. Two types of errors can occur. The errors of the first type are detected in the code of the source and signaled by an "exception" in C++. The serious errors (second type) result in a signal that makes them apparent. In the first case, the exception can be handled at any time, even partially. Two cases can be distinguished by the source of the exception. If the exception has been raised during the agent's reaction, the agent is marked as having failed, the operations in the database are cancelled and the message queue is re-stored. The execution engine (1) can process the next notification. If not, the exception either represents an expected signal, in which case the corresponding notification is created and sent, or it is id the occurrence of a serious error. In this case, the processing of an error signal depends on the position in the loop at which it has been signaled. If the error occurs during the agent's reaction to a notification, then it is processed in the same way as an exception would be at this moment. If the error results from the analysis of the notification or from the loading of the target agent, then the notification is ignored.

The algorithm (6) of the agent machine manages the distribution of the notifications to all the entities in the network, and the reliability of these operations, especially their scheduling. It ensures that the messages have actually been delivered to the destination agents in the order in which they were sent. The relationship between the notifications is established simply by numbering the messages, by incrementing a logical clock. A logical clock makes it possible to obtain a certain number of properties (the causality property of the channel, for example).

A matrix clock is included in each message, and each time a message is received, a part of the algorithm will first read the position of the clock.

A message is an object that contains four pieces of information: the identifier of the sending agent, the identifier of the destination agent, a clock, and the notification. If there are n sites, the clock is composed of a matrix nxn, since the status of the communication of each site with each other site must be maintained. Each machine will try to maintain its own view of the global communications, even the communications between two other machines. A matrix nxn corresponds to a very high-volume piece of information. An optimization of the transferred data is performed. A part of the sending algorithm calculates the part of the clock actually used and transfers only this part.

Thus, the invention provides the agent machine with the property of failure resistance, since a large process is divided into small elementary processes. In particular, it applies to Data Warehousing and to the distribution of the process. The data transforming and routing process is broken down into elementary steps. In a distributed process, several machines are impacted and the risk of a machine or network failure is high. Breaking the process down into steps makes it possible to isolate one piece in particular, i.e., the transmission of the notification from one machine to another.

The agent machine is distributed. The agent machines communicate with one another; the agents do not see this distribution since they communicate via notifications and address their local agent machine. An agent's reaction remains local. The atomic reaction, which is to change the status and to send other notifications, remains local. Thus, none of the network failures have any influence on this reaction. Moreover, the transmission of the notifications itself is an atomic operation. Each notification sent is an atomic operation, which makes it possible to handle network failures by saving the notifications while waiting for the machine or the network to return to normal operation. Transmission errors, site failures and network failures are handled at the agent machine level.

A notification is something asynchronous. When a notification is sent, the program can continue in an asynchronous mode. When the notification is sent, it is certain that it will be received and that the consequence of the message that has been sent will be executed, and there is no need to wait for the exact result. This corresponds to the property of atomicity in the delivery of the messages and in the order of delivery of the messages. With the rescheduling of the messages in causal order, and with the reliability of the delivery of the messages, the sender can completely rely on the properties provided, so as not to have to re-synchronize in order to be sure that the operations will be able to be executed in the correct order. Thus, the handling of data errors is eliminated from the program.

Guaranteeing this reliability of the network, and this scheduling at the agent machine level, makes it possible to extract a relatively simple status graph, which is easy to handle since includes only the applicable errors. The event level (Agent Event) provides something of a structure for the various statuses passed through, and makes it possible, in particular, to handle the errors that are currently applicable at this level, which can occur at each step. The applicable errors are handled by the status automation. By breaking things down into elementary operations (transfer step, loading step), and by having a controller that reacts correctly, for which there is a certainty of its reacting correctly, it is possible to identify the step and to restart this step only.

The agent Event is an agent specific to Data Warehousing. The agent machine according to the invention can therefore be used in the Data Warehousing field by manipulating this specific agent as defined in the invention. Of course, the agent machine can be used in other fields, using agents specific to each of these fields. For example, the agent machine can be used in the firewall field (Netwall). A firewall authorizes or inhibits connections to a network, and all the information related to the actions it executes are placed in a log, a file (Log) whose size increases rapidly. The process according to the invention can be rewritten in another language, and can be used to manage this file (Log) using agents specific to the firewall field (NetwallEvent). These specific agents use the same model for changing the status of the agent "Event" specific to Data Warehousing, but are adapted for managing data of the file type specific to the application of the fire wall.

While this invention has been described in conjunction with specific embodiments thereof, it is evident that many alternatives, modifications and variations will be apparent to those skilled in the art. Accordingly, the preferred embodiments of the invention as set forth herein, are intended to be illustrative, not limiting. Various changes may be made without departing from the spirit and scope of the invention as set forth herein and defined in the claims.

What is claimed is:

1. A process for transforming and routing data between agent servers in one or more of a plurality of first machines and a central agent server of a second machine, wherein an agent server comprises autonomous agents that communicate via messages, each message having at least a notification, and wherein each agent belongs to one type of agent and is designated by an agent identifier, the agents being shared out into at least two types, with each type of agent handling only one type of notification, the agent server also comprising a storage layer and an agent machine having an execution engine, a communication channel and two message queues associated with the notifications, namely, a local queue and an external queue, the process comprising taking a notification from the local queue, by the execution engine, determining and loading a corresponding agent by its agent identifier for each notification, depending on which type of agent is required to handle the notification, causing the agent to react to the notification, such that the agent can then change its status and send addressed notifications to the communication channel, storing the notifications in the local queue if said notifications are addressed to a local agent and in the external queue if said notifications are addressed to an agent of another server.

2. A process for transforming and routing data according to claim 1, characterized in that the execution engine of the agent machine executes, after the agent's reaction, a save of the agent's status, a deletion of the processed notification from the local message queue, and atomically, a validation of various operations performed that have changed either the status of the agent or the status of the message queues.

3. A process for transforming and routing data according to claim 1, characterized in that each message includes an optimized matrix clock that can be read by the execution engine of the agent machine, thus making it possible to process the notifications in their causal sending order.

4. A process for transforming and routing data according to claim 2, characterized in that each message includes an optimized matrix clock that can be read by the execution engine of the agent machine, thus making it possible to process the notifications in their causal sending order.

5. A process for transforming and routing data according to claim 1, characterized in that a message is an object that contains four pieces of information, including an identifier of the sending agent, an identifier of the destination agent, a clock and the notification.

6. A process for transforming and routing data according to claim 1, characterized in that the agent machine manages

distribution to all entities of the network, reliability of operations, and scheduling of messages which is established by incrementing a logical matrix clock.

7. A process for transforming and routing data according to claim 1, characterized in that the communication channel performs the following separate transactions:

storing a notification sent by an agent locally, either in the local queue for notifications addressed to a local agent, or in the external queue for an agent located in another server;

at the end of the agent's reaction, saving the two queues on a disk, together with the reacting agent;

extracting by a first function of the network each notification, in turn, from the external queue and sending the extracted notification to a targeted agent server;

receiving the notifications from the other servers by a second function of the network and inserting the notifications into the local queue in causal order.

8. A process for transforming and routing data according to claim 1, characterized in that when a targeted agent server receives a message, the targeted agent server stores the message in a persistent storage layer, validates the operation and confirms the reception of the message to a source server; and upon reception of this confirmation, the source server deletes the message from the external queue of the source server.

9. A process for transforming and routing data according to claim 1, characterized in that a persistent external service ensures the atomicity of the operations that have changed either the status of the agent or the status of the message queues.

10. A process for transforming and routing data according to claim 1, characterized in that the execution engine includes a program that:

takes a notification that is in the local queue, using a Get function;

identifies which agent the notification is addressed to;

loads the identified agent corresponding to the notification;

causes the agent to react by calling a reaction operation such that the agent can change its status by modifying its image in memory, and can send notifications by addressing the channel that will distribute the notification, in which case the notifications are routed to a message queue where they are stored in chronological order;

if the reaction operation is executed, saves the status of the agent, which has been changed by the reaction it has executed, deletes the notification that has been processed, and validates all of the operations that have been performed; and

starts again in the program loop in order to process another notification.

11. A process for transforming and routing data according to claim 1, characterized in that the process is executed between agent servers present in machines of production sites and a central agent server present in a machine of a decision-making base, and uses a particular agent that describes specific events, in which a status controller is implemented.

12. A process for transforming and routing data according to claim 11, characterized in that the status controller comprises various statuses including:

a None status in which the event is not active;

a notification status that causes the event to pass into a Wait status in which the event waits for its activation condition to become "true"; at which time the event delivers itself a notification that changes the event to an Init status;

a first simple event status that attains a Run status directly and executes its action;

a first composed event status that indicates to sub-events agreement for the sub-event to start and waits for the first of the sub-events to attain a Run status in order for the composed events to pass into the Run status;

a second simple event which then passes into a Done or Fail status, depending on the status of the simple event action;

a second composed event, which passes in the Done status on the execution statuses of all the sub-events;

an event in execution status that can be killed by sending to the event in execution status a notification, in which case the event passes into a Kill status, marking the fact that the event has been killed, and then to a Stop status when it has actually been killed;

a terminated event, i.e., in the Done, Fail or Stop status that can be re-executed by a client by sending the event a notification, in which case the event is reset and passes into the Wait status; the event then propagates the notification to all the sub-events; an event executed repeatedly and endowed with a Done status is automatically reset by an agent Scheduler, which sends the event a notification; and

an event in the Fail or Stop status can be re-executed without being reset by sending a notification, and when this condition becomes "true," the event restarts at the point at which it was previously stopped.

13. A process for transforming and routing data according to claim 12, further comprising adding a new function by adding an agent class derived from the class "Event."

14. A process for transforming and routing data according to claim 11, characterized in that three categories of "Event" type agents—"SeqEvent," "ParEvent" and "complexEvent"—take on elementary operations of the "Event" type in order to respectively execute them in sequence, in parallel, or in accordance with the dependencies between the various operations.

* * * * *